NAME

CGI − Simple Common Gateway Interface Class

ABSTRACT

This perl library uses perl5 objects to make it easy to create Web fill-out forms and parse their contents. This package defines CGI objects, entities that contain the values of the current query string and other state variables. Using a CGI object's methods, you can examine keywords and parameters passed to your script, and create forms whose initial values are taken from the current query (thereby preserving state information).

The current version of CGI.pm is available at

```
http://www.genome.wi.mit.edu/ftp/pub/software/WWW/cgi_docs.html
ftp://ftp-genome.wi.mit.edu/pub/software/WWW/
```

INSTALLATION:

To install this package, just change to the directory in which this file is found and type the following:

```
perl Makefile.PL
make
make install
```

This will copy CGI.pm to your perl library directory for use by all perl scripts. You probably must be root to do this. Now you can load the CGI routines in your Perl scripts with the line:

```
use CGI;
```

If you don't have sufficient privileges to install CGI.pm in the Perl library directory, you can put CGI.pm into some convenient spot, such as your home directory, or in cgi-bin itself and prefix all Perl scripts that call it with something along the lines of the following preamble:

```
use lib '/home/davis/lib';
use CGI;
```

If you are using a version of perl earlier than 5.002 (such as NT perl), use this instead:

```
BEGIN {
        unshift(@INC,'/home/davis/lib');
}
use CGI;
```

The CGI distribution also comes with a cute module called the *CGI::Carp* manpage. It redefines the *die()*, *warn()*, *confess()* and *croak()* error routines so that they write nicely formatted error messages into the server's error log (or to the output stream of your choice). This avoids long hours of groping through the error and access logs, trying to figure out which CGI script is generating error messages. If you choose, you can even have fatal error messages echoed to the browser to avoid the annoying and uninformative "Server Error" message.

DESCRIPTION

CREATING A NEW QUERY OBJECT:

```
$query = new CGI;
```

This will parse the input (from both POST and GET methods) and store it into a perl5 object called `$query`.

CREATING A NEW QUERY OBJECT FROM AN INPUT FILE

```
$query = new CGI(INPUTFILE);
```

If you provide a file handle to the *new()* method, it will read parameters from the file (or STDIN, or whatever). The file can be in any of the forms describing below under debugging (i.e. a series of newline delimited TAG=VALUE pairs will work). Conveniently, this type of file is created by the *save()* method (see below). Multiple records can be saved and restored.

Perl purists will be pleased to know that this syntax accepts references to file handles, or even references to filehandle globs, which is the "official" way to pass a filehandle:

```
$query = new CGI(\*STDIN);
```

You can also initialize the query object from an associative array reference:

```
$query = new CGI( {'dinosaur'=>'barney',
                   'song'=>'I love you',
                   'friends'=>[qw/Jessica George Nancy/]}
                );
```

or from a properly formatted, URL–escaped query string:

```
$query = new CGI('dinosaur=barney&color=purple');
```

To create an empty query, initialize it from an empty string or hash:

```
    $empty_query = new CGI("");
         -or-
    $empty_query = new CGI({});
```

FETCHING A LIST OF KEYWORDS FROM THE QUERY:

```
@keywords = $query->keywords
```

If the script was invoked as the result of an <ISINDEX> search, the parsed keywords can be obtained as an array using the *keywords()* method.

FETCHING THE NAMES OF ALL THE PARAMETERS PASSED TO YOUR SCRIPT:

```
@names = $query->param
```

If the script was invoked with a parameter list (e.g. "name1=value1&name2=value2&name3=value3"), the *param()* method will return the parameter names as a list. If the script was invoked as an <ISINDEX> script, there will be a single parameter named 'keywords'.

NOTE: As of version 1.5, the array of parameter names returned will be in the same order as they were submitted by the browser. Usually this order is the same as the order in which the parameters are defined in the form (however, this isn't part of the spec, and so isn't guaranteed).

FETCHING THE VALUE OR VALUES OF A SINGLE NAMED PARAMETER:

```
@values = $query->param('foo');

          -or-

$value = $query->param('foo');
```

Pass the *param()* method a single argument to fetch the value of the named parameter. If the parameter is multivalued (e.g. from multiple selections in a scrolling list), you can ask to receive an array. Otherwise the method will return a single value.

SETTING THE *VALUE* (S) OF A NAMED PARAMETER:

```
$query->param('foo','an','array','of','values');
```

This sets the value for the named parameter 'foo' to an array of values. This is one way to change the value of a field AFTER the script has been invoked once before. (Another way is with the −override parameter accepted by all methods that generate form elements.)

param() also recognizes a named parameter style of calling described in more detail later:

```
$query->param(-name=>'foo',-values=>['an','array','of','values']);

                              -or-

$query->param(-name=>'foo',-value=>'the value');
```

APPENDING ADDITIONAL VALUES TO A NAMED PARAMETER:

```
$query->append(-name=>;'foo',-values=>['yet','more','values']);
```

This adds a value or list of values to the named parameter. The values are appended to the end of the parameter if it already exists. Otherwise the parameter is created. Note that this method only recognizes the named argument calling syntax.

IMPORTING ALL PARAMETERS INTO A NAMESPACE:

```
$query->import_names('R');
```

This creates a series of variables in the 'R' namespace. For example, `$R::foo`, `@R:foo`. For keyword lists, a variable `@R::keywords` will appear. If no namespace is given, this method will assume 'Q'. WARNING: don't import anything into 'main'; this is a major security risk!!!!

In older versions, this method was called **import()**. As of version 2.20, this name has been removed completely to avoid conflict with the built-in Perl module **import** operator.

DELETING A PARAMETER COMPLETELY:

```
$query->delete('foo');
```

This completely clears a parameter. It sometimes useful for resetting parameters that you don't want passed down between script invocations.

DELETING ALL PARAMETERS:

$query->*delete_all()*;

This clears the CGI object completely. It might be useful to ensure that all the defaults are taken when you create a fill-out form.

SAVING THE STATE OF THE FORM TO A FILE:

```
$query->save(FILEHANDLE)
```

This will write the current state of the form to the provided filehandle. You can read it back in by providing a filehandle to the *new()* method. Note that the filehandle can be a file, a pipe, or whatever!

The format of the saved file is:

```
NAME1=VALUE1
NAME1=VALUE1'
NAME2=VALUE2
NAME3=VALUE3
=
```

Both name and value are URL escaped. Multi-valued CGI parameters are represented as repeated names. A session record is delimited by a single = symbol. You can write out multiple records and read them back in with several calls to **new**. You can do this across several sessions by opening the file in append mode, allowing you to create primitive guest books, or to keep a history of users' queries. Here's a short example of creating multiple session records:

```
use CGI;

open (OUT,">>test.out") || die;
$records = 5;
foreach (0..$records) {
    my $q = new CGI;
    $q->param(-name=>'counter',-value=>$_);
    $q->save(OUT);
}
close OUT;

# reopen for reading
open (IN,"test.out") || die;
while (!eof(IN)) {
    my $q = new CGI(IN);
    print $q->param('counter'),"\n";
}
```

The file format used for save/restore is identical to that used by the Whitehead Genome Center's data exchange format "Boulderio", and can be manipulated and even databased using Boulderio utilities. See

http://www.genome.wi.mit.edu/genome_software/other/boulder.html

for further details.

CREATING A SELF-REFERENCING URL THAT PRESERVES STATE INFORMATION:

```
$myself = $query->self_url;
print "<A HREF=$myself>I'm talking to myself.</A>";
```

self_url() will return a URL, that, when selected, will reinvoke this script with all its state information intact. This is most useful when you want to jump around within the document using internal anchors but you don't want to disrupt the current contents of the *form*(s). Something like this will do the trick.

```
$myself = $query->self_url;
print "<A HREF=$myself#table1>See table 1</A>";
print "<A HREF=$myself#table2>See table 2</A>";
print "<A HREF=$myself#yourself>See for yourself</A>";
```

If you don't want to get the whole query string, call the method *url()* to return just the URL for the script:

```
$myself = $query->url;
print "<A HREF=$myself>No query string in this baby!</A>\n";
```

You can also retrieve the unprocessed query string with *query_string()*:

```
$the_string = $query->query_string;
```

COMPATIBILITY WITH CGI–LIB.PL

To make it easier to port existing programs that use cgi-lib.pl the compatibility routine "ReadParse" is provided. Porting is simple:

OLD VERSION
  require "cgi-lib.pl";
  &ReadParse;
  print "The value of the antique is `$in`{antique}.\n";

NEW VERSION
  use CGI;
  CGI::ReadParse
  print "The value of the antique is `$in`{antique}.\n";

CGI.pm's *ReadParse()* routine creates a tied variable named `%in`, which can be accessed to obtain the query variables. Like ReadParse, you can also provide your own variable. Infrequently used features of ReadParse, such as the creation of `@in` and `$in` variables, are not supported.

Once you use ReadParse, you can retrieve the query object itself this way:

```
$q = $in{CGI};
print $q->textfield(-name=>'wow',
                    -value=>'does this really work?');
```

This allows you to start using the more interesting features of CGI.pm without rewriting your old scripts from scratch.

CALLING CGI FUNCTIONS THAT TAKE MULTIPLE ARGUMENTS

In versions of CGI.pm prior to 2.0, it could get difficult to remember the proper order of arguments in CGI function calls that accepted five or six different arguments. As of 2.0, there's a better way to pass arguments to the various CGI functions. In this style, you pass a series of name=>argument pairs, like this:

```
$field = $query->radio_group(-name=>'OS',
                             -values=>[Unix,Windows,Macintosh],
                             -default=>'Unix');
```

The advantages of this style are that you don't have to remember the exact order of the arguments, and if you leave out a parameter, in most cases it will default to some reasonable value. If you provide a parameter that the method doesn't recognize, it will usually do something useful with it, such as incorporating it into the HTML form tag. For example if Netscape decides next week to add a new JUSTIFICATION parameter to the text field tags, you can start using the feature without waiting for a new version of CGI.pm:

```
$field = $query->textfield(-name=>'State',
                           -default=>'gaseous',
                           -justification=>'RIGHT');
```

This will result in an HTML tag that looks like this:

```
<INPUT TYPE="textfield" NAME="State" VALUE="gaseous"
       JUSTIFICATION="RIGHT">
```

Parameter names are case insensitive: you can use –name, or –Name or –NAME. You don't have to use the hyphen if you don't want to. After creating a CGI object, call the **use_named_parameters()** method with a nonzero value. This will tell CGI.pm that you intend to use named parameters exclusively:

```
$query = new CGI;
$query->use_named_parameters(1);
$field = $query->radio_group('name'=>'OS',
                             'values'=>['Unix','Windows','Macintosh'],
                             'default'=>'Unix');
```

Actually, CGI.pm only looks for a hyphen in the first parameter. So you can leave it off subsequent parameters if you like. Something to be wary of is the potential that a string constant like "values" will collide with a keyword (and in fact it does!) While Perl usually figures out when you're referring to a function and when you're referring to a string, you probably should put quotation marks around all string constants just to play it safe.

CREATING THE HTTP HEADER:

```
print $query->header;

     -or-

print $query->header('image/gif');

     -or-

print $query->header('text/html','204 No response');

     -or-

print $query->header(-type=>'image/gif',
                     -nph=>1,
                     -status=>'402 Payment required',
                     -expires=>'+3d',
                     -cookie=>$cookie,
                     -Cost=>'$2.00');
```

header() returns the Content-type: header. You can provide your own MIME type if you choose, otherwise it defaults to text/html. An optional second parameter specifies the status code and a human-readable message. For example, you can specify 204, "No response" to create a script that tells the browser to do nothing at all. If you want to add additional fields to the header, just tack them on to the end:

```
print $query->header('text/html','200 OK','Content-Length: 3002');
```

The last example shows the named argument style for passing arguments to the CGI methods using named parameters. Recognized parameters are **–type**, **–status**, **–expires**, and **–cookie**. Any other parameters will be stripped of their initial hyphens and turned into header fields, allowing you to specify any HTTP header you desire.

Most browsers will not cache the output from CGI scripts. Every time the browser reloads the page, the script is invoked anew. You can change this behavior with the **–expires** parameter. When you specify an absolute or relative expiration interval with this parameter, some browsers and proxy servers will cache the script's output until the indicated expiration date. The following forms are all valid for the –expires field:

```
+30s                              30 seconds from now
+10m                              ten minutes from now
+1h                               one hour from now
-1d                               yesterday (i.e. "ASAP!")
now                               immediately
+3M                               in three months
+10y                              in ten years time
Thursday, 25-Apr-96 00:40:33 GMT  at the indicated time & date
```

(*CGI::expires()* is the static function call used internally that turns relative time intervals into HTTP dates. You can call it directly if you wish.)

The **–cookie** parameter generates a header that tells the browser to provide a "magic cookie" during all subsequent transactions with your script. Netscape cookies have a special format that includes interesting attributes such as expiration time. Use the *cookie()* method to create and retrieve session cookies.

The **–nph** parameter, if set to a true value, will issue the correct headers to work with a NPH (no-parse-header) script. This is important to use with certain servers, such as Microsoft Internet Explorer, which expect all their scripts to be NPH.

GENERATING A REDIRECTION INSTRUCTION

```
print $query->redirect('http://somewhere.else/in/movie/land');
```

redirects the browser elsewhere. If you use redirection like this, you should **not** print out a header as well. As of version 2.0, we produce both the unofficial Location: header and the official URI: header. This should satisfy most servers and browsers.

One hint I can offer is that relative links may not work correctly when when you generate a redirection to another document on your site. This is due to a well-intentioned optimization that some servers use. The solution to this is to use the full URL (including the http: part) of the document you are redirecting to.

You can use named parameters:

```
print $query->redirect(-uri=>'http://somewhere.else/in/movie/land',
                       -nph=>1);
```

The **–nph** parameter, if set to a true value, will issue the correct headers to work with a NPH (no-parse-header) script. This is important to use with certain servers, such as Microsoft Internet Explorer, which expect all their scripts to be NPH.

CREATING THE HTML HEADER:

```
print $query->start_html(-title=>'Secrets of the Pyramids',
                         -author=>'fred@capricorn.org',
                         -base=>'true',
                         -target=>'_blank',
                         -meta=>{'keywords'=>'pharaoh secret mummy',
                                 'copyright'=>'copyright 1996 King Tut'},
                         -BGCOLOR=>'blue');

-or-
```

```
print $query->start_html('Secrets of the Pyramids',
                         'fred@capricorn.org','true',
                         'BGCOLOR="blue"');
```

This will return a canned HTML header and the opening <BODY> tag. All parameters are optional. In the named parameter form, recognized parameters are −title, −author, −base, −xbase and −target (see below for the explanation). Any additional parameters you provide, such as the Netscape unofficial BGCOLOR attribute, are added to the <BODY> tag.

The argument **−xbase** allows you to provide an HREF for the <BASE> tag different from the current location, as in

```
-xbase=>"http://home.mcom.com/"
```

All relative links will be interpreted relative to this tag.

The argument **−target** allows you to provide a default target frame for all the links and fill-out forms on the page. See the Netscape documentation on frames for details of how to manipulate this.

```
-target=>"answer_window"
```

All relative links will be interpreted relative to this tag. You add arbitrary meta information to the header with the **−meta** argument. This argument expects a reference to an associative array containing name/value pairs of meta information. These will be turned into a series of header <META> tags that look something like this:

```
<META NAME="keywords" CONTENT="pharaoh secret mummy">
<META NAME="description" CONTENT="copyright 1996 King Tut">
```

There is no support for the HTTP−EQUIV type of <META> tag. This is because you can modify the HTTP header directly with the **header()** method.

JAVASCRIPTING: The **−script**, **−onLoad** and **−onUnload** parameters are used to add Netscape JavaScript calls to your pages. **−script** should point to a block of text containing JavaScript function definitions. This block will be placed within a <SCRIPT> block inside the HTML (not HTTP) header. The block is placed in the header in order to give your page a fighting chance of having all its JavaScript functions in place even if the user presses the stop button before the page has loaded completely. CGI.pm attempts to format the script in such a way that JavaScript-naive browsers will not choke on the code: unfortunately there are some browsers, such as Chimera for Unix, that get confused by it nevertheless.

The **−onLoad** and **−onUnload** parameters point to fragments of JavaScript code to execute when the page is respectively opened and closed by the browser. Usually these parameters are calls to functions defined in the **−script** field:

```
$query = new CGI;
print $query->header;
$JSCRIPT=<<END;
// Ask a silly question
function riddle_me_this() {
   var r = prompt("What walks on four legs in the morning, " +
                  "two legs in the afternoon, " +
                  "and three legs in the evening?");
   response(r);
}
// Get a silly answer
function response(answer) {
   if (answer == "man")
      alert("Right you are!");
   else
      alert("Wrong!  Guess again.");
}
END
print $query->start_html(-title=>'The Riddle of the Sphinx',
                         -script=>$JSCRIPT);
```

See

```
http://home.netscape.com/eng/mozilla/2.0/handbook/javascript/
```

for more information about JavaScript.

The old-style positional parameters are as follows:

Parameters:

1. The title
2. The author's e-mail address (will create a <LINK REV="MADE"> tag if present
3. A 'true' flag if you want to include a <BASE> tag in the header. This helps resolve relative addresses to absolute ones when the document is moved, but makes the document hierarchy non-portable. Use with care!

4, 5, 6...
Any other parameters you want to include in the <BODY> tag. This is a good place to put Netscape extensions, such as colors and wallpaper patterns.

ENDING THE HTML DOCUMENT:

```
print $query->end_html
```

This ends an HTML document by printing the </BODY></HTML> tags.

CREATING FORMS:

General note The various form-creating methods all return strings to the caller, containing the tag or tags that will create the requested form element. You are responsible for actually printing out these strings. It's set up this way so that you can place formatting tags around the form elements.

Another note The default values that you specify for the forms are only used the **first** time the script is invoked (when there is no query string). On subsequent invocations of the script (when there is a query string), the former values are used even if they are blank.

If you want to change the value of a field from its previous value, you have two choices:

(1) call the *param()* method to set it.

(2) use the −override (alias −force) parameter (a new feature in version 2.15). This forces the default value to be used, regardless of the previous value:

```
print $query->textfield(-name=>'field_name',
                        -default=>'starting value',
                        -override=>1,
                        -size=>50,
                        -maxlength=>80);
```

Yet another note By default, the text and labels of form elements are escaped according to HTML rules. This means that you can safely use "<CLICK ME>" as the label for a button. However, it also interferes with your ability to incorporate special HTML character sequences, such as Á, into your fields. If you wish to turn off automatic escaping, call the *autoEscape()* method with a false value immediately after creating the CGI object:

```
$query = new CGI;
$query->autoEscape(undef);
```

CREATING AN ISINDEX TAG

```
print $query->isindex(-action=>$action);

      -or-

print $query->isindex($action);
```

Prints out an <ISINDEX> tag. Not very exciting. The parameter −action specifies the URL of the script to process the query. The default is to process the query with the current script.

STARTING AND ENDING A FORM

```
print $query->startform(-method=>$method,
                        -action=>$action,
                        -encoding=>$encoding);
  <... various form stuff ...>
print $query->endform;

    -or-

print $query->startform($method,$action,$encoding);
  <... various form stuff ...>
print $query->endform;
```

startform() will return a <FORM> tag with the optional method, action and form encoding that you specify. The defaults are:
method: POST
action: this script
encoding: application/x-www-form-urlencoded

endform() returns the closing </FORM> tag.

Startform()'s encoding method tells the browser how to package the various fields of the form before sending the form to the server. Two values are possible:

application/x-www-form-urlencoded

This is the older type of encoding used by all browsers prior to Netscape 2.0. It is compatible with many CGI scripts and is suitable for short fields containing text data. For your convenience, CGI.pm stores the name of this encoding type in **$CGI::URL_ENCODED**.

multipart/form-data

This is the newer type of encoding introduced by Netscape 2.0. It is suitable for forms that contain very large fields or that are intended for transferring binary data. Most importantly, it enables the "file upload" feature of Netscape 2.0 forms. For your convenience, CGI.pm stores the name of this encoding type in **$CGI::MULTIPART**

Forms that use this type of encoding are not easily interpreted by CGI scripts unless they use CGI.pm or another library designed to handle them.

For compatibility, the *startform()* method uses the older form of encoding by default. If you want to use the newer form of encoding by default, you can call **start_multipart_form()** instead of **startform()**.

JAVASCRIPTING: The **−name** and **−onSubmit** parameters are provided for use with JavaScript. The −name parameter gives the form a name so that it can be identified and manipulated by JavaScript functions. −onSubmit should point to a JavaScript function that will be executed just before the form is submitted to your server. You can use this opportunity to check the contents of the form for consistency and completeness. If you find something wrong, you can put up an alert box or maybe fix things up yourself. You can abort the submission by returning false from this function.

Usually the bulk of JavaScript functions are defined in a <SCRIPT> block in the HTML header and −onSubmit points to one of these function call. See *start_html()* for details.

CREATING A TEXT FIELD

```
print $query->textfield(-name=>'field_name',
                        -default=>'starting value',
                        -size=>50,
                        -maxlength=>80);
    -or-

print $query->textfield('field_name','starting value',50,80);
```

textfield() will return a text input field.

Parameters

1. The first parameter is the required name for the field (−name).
2. The optional second parameter is the default starting value for the field contents (−default).
3. The optional third parameter is the size of the field in
   characters (−size).
4. The optional fourth parameter is the maximum number of characters the
   field will accept (−maxlength).

As with all these methods, the field will be initialized with its previous contents from earlier invocations of the script. When the form is processed, the value of the text field can be retrieved with:

```
$value = $query->param('foo');
```

If you want to reset it from its initial value after the script has been called once, you can do so like this:

```
$query->param('foo',"I'm taking over this value!");
```

NEW AS OF VERSION 2.15: If you don't want the field to take on its previous value, you can force its current value by using the −override (alias −force) parameter:

```
 print $query->textfield(-name=>'field_name',
                         -default=>'starting value',
                         -override=>1,
                         -size=>50,
                         -maxlength=>80);
```

JAVASCRIPTING: You can also provide **–onChange**, **–onFocus**, **–onBlur** and **–onSelect** parameters to register JavaScript event handlers. The onChange handler will be called whenever the user changes the contents of the text field. You can do text validation if you like. onFocus and onBlur are called respectively when the insertion point moves into and out of the text field. onSelect is called when the user changes the portion of the text that is selected.

CREATING A BIG TEXT FIELD

```
print $query->textarea(-name=>'foo',
                       -default=>'starting value',
                       -rows=>10,
                       -columns=>50);

     -or

print $query->textarea('foo','starting value',10,50);
```

textarea() is just like textfield, but it allows you to specify rows and columns for a multiline text entry box. You can provide a starting value for the field, which can be long and contain multiple lines.

JAVASCRIPTING: The **–onChange**, **–onFocus**, **–onBlur** and **–onSelect** parameters are recognized. See *textfield()*.

CREATING A PASSWORD FIELD

```
print $query->password_field(-name=>'secret',
                             -value=>'starting value',
                             -size=>50,
                             -maxlength=>80);
     -or-

print $query->password_field('secret','starting value',50,80);
```

password_field() is identical to *textfield()*, except that its contents will be starred out on the web page.

JAVASCRIPTING: The **–onChange**, **–onFocus**, **–onBlur** and **–onSelect** parameters are recognized. See *textfield()*.

CREATING A FILE UPLOAD FIELD

```
 print $query->filefield(-name=>'uploaded_file',
                         -default=>'starting value',
                         -size=>50,
                         -maxlength=>80);
     -or-

 print $query->filefield('uploaded_file','starting value',50,80);
```

filefield() will return a file upload field for Netscape 2.0 browsers. In order to take full advantage of this *you must use the new multipart encoding scheme* for the form. You can do this either by calling **startform()** with an encoding type of **$CGI::MULTIPART**, or by calling the new method

start_multipart_form() instead of vanilla **startform()**.

Parameters

1. The first parameter is the required name for the field (−name).
2. The optional second parameter is the starting value for the field contents to be used as the default file name (−default).

   The beta2 version of Netscape 2.0 currently doesn't pay any attention to this field, and so the starting value will always be blank. Worse, the field loses its "sticky" behavior and forgets its previous contents. The starting value field is called for in the HTML specification, however, and possibly later versions of Netscape will honor it.
3. The optional third parameter is the size of the field in characters (−size).
4. The optional fourth parameter is the maximum number of characters the field will accept (−maxlength).

When the form is processed, you can retrieve the entered filename by calling *param()*.

```
$filename = $query->param('uploaded_file');
```

In Netscape Gold, the filename that gets returned is the full local filename on the **remote user's** machine. If the remote user is on a Unix machine, the filename will follow Unix conventions:

```
/path/to/the/file
```

On an MS−DOS/Windows and OS/2 machines, the filename will follow DOS conventions:

```
C:\PATH\TO\THE\FILE.MSW
```

On a Macintosh machine, the filename will follow Mac conventions:

```
HD 40:Desktop Folder:Sort Through:Reminders
```

The filename returned is also a file handle. You can read the contents of the file using standard Perl file reading calls:

```
# Read a text file and print it out
while (<$filename>) {
   print;
}

# Copy a binary file to somewhere safe
open (OUTFILE,">>/usr/local/web/users/feedback");
while ($bytesread=read($filename,$buffer,1024)) {
   print OUTFILE $buffer;
}
```

When a file is uploaded the browser usually sends along some information along with it in the format of headers. The information usually includes the MIME content type. Future browsers may send other information as well (such as modification date and size). To retrieve this information, call *uploadInfo()*. It returns a reference to an associative array containing all the document headers.

```
$filename = $query->param('uploaded_file');
$type = $query->uploadInfo($filename)->{'Content-Type'};
unless ($type eq 'text/html') {
   die "HTML FILES ONLY!";
}
```

If you are using a machine that recognizes "text" and "binary" data modes, be sure to understand when and how to use them (see the Camel book). Otherwise you may find that binary files are corrupted during file uploads.

JAVASCRIPTING: The **−onChange**, **−onFocus**, **−onBlur** and **−onSelect** parameters are recognized. See *textfield()* for details.

CREATING A POPUP MENU

```
print $query->popup_menu('menu_name',
                         ['eenie','meenie','minie'],
                         'meenie');

   -or-

%labels = ('eenie'=>'your first choice',
           'meenie'=>'your second choice',
           'minie'=>'your third choice');
print $query->popup_menu('menu_name',
                         ['eenie','meenie','minie'],
                         'meenie',\%labels);

     -or (named parameter style)-

print $query->popup_menu(-name=>'menu_name',
                         -values=>['eenie','meenie','minie'],
                         -default=>'meenie',
                         -labels=>\%labels);
```

popup_menu() creates a menu.

1. The required first argument is the menu's name (−name).
2. The required second argument (−values) is an array **reference** containing the list of menu items in the menu. You can pass the method an anonymous array, as shown in the example, or a reference to a named array, such as "\@foo".
3. The optional third parameter (−default) is the name of the default menu choice. If not specified, the first item will be the default. The values of the previous choice will be maintained across queries.
4. The optional fourth parameter (−labels) is provided for people who want to use different values for the user-visible label inside the popup menu nd the value returned to your script. It's a pointer to an associative array relating menu values to user-visible labels. If you leave this parameter blank, the menu values will be displayed by default. (You can also leave a label undefined if you want to).

When the form is processed, the selected value of the popup menu can be retrieved using:

```
$popup_menu_value = $query->param('menu_name');
```

JAVASCRIPTING: *popup_menu()* recognizes the following event handlers: **−onChange**, **−onFocus**, and **−onBlur**. See the *textfield()* section for details on when these handlers are called.

CREATING A SCROLLING LIST

```
print $query->scrolling_list('list_name',
                             ['eenie','meenie','minie','moe'],
                             ['eenie','moe'],5,'true');
   -or-
```

```
print $query->scrolling_list('list_name',
                             ['eenie','meenie','minie','moe'],
                             ['eenie','moe'],5,'true',
                             \%labels);

      -or-

print $query->scrolling_list(-name=>'list_name',
                             -values=>['eenie','meenie','minie','moe'],
                             -default=>['eenie','moe'],
                             -size=>5,
                             -multiple=>'true',
                             -labels=>\%labels);
```

scrolling_list() creates a scrolling list.

Parameters:

1. The first and second arguments are the list name (−name) and values (−values). As in the popup menu, the second argument should be an array reference.
2. The optional third argument (−default) can be either a reference to a list containing the values to be selected by default, or can be a single value to select. If this argument is missing or undefined, then nothing is selected when the list first appears. In the named parameter version, you can use the synonym "−defaults" for this parameter.
3. The optional fourth argument is the size of the list (−size).
4. The optional fifth argument can be set to true to allow multiple simultaneous selections (−multiple). Otherwise only one selection will be allowed at a time.
5. The optional sixth argument is a pointer to an associative array containing long user-visible labels for the list items (−labels). If not provided, the values will be displayed.

   When this form is processed, all selected list items will be returned as a list under the parameter name 'list_name'. The values of the selected items can be retrieved with:

   ```
   @selected = $query->param('list_name');
   ```

JAVASCRIPTING: *scrolling_list()* recognizes the following event handlers: **−onChange**, **−onFocus**, and **−onBlur**. See *textfield()* for the description of when these handlers are called.

CREATING A GROUP OF RELATED CHECKBOXES

```
print $query->checkbox_group(-name=>'group_name',
                             -values=>['eenie','meenie','minie','moe'],
                             -default=>['eenie','moe'],
                             -linebreak=>'true',
                             -labels=>\%labels);

print $query->checkbox_group('group_name',
                             ['eenie','meenie','minie','moe'],
                             ['eenie','moe'],'true',\%labels);

HTML3-COMPATIBLE BROWSERS ONLY:
```

```
print $query->checkbox_group(-name=>'group_name',
                             -values=>['eenie','meenie','minie','moe'],
                             -rows=2,-columns=>2);
```

checkbox_group() creates a list of checkboxes that are related by the same name.

Parameters:

1. The first and second arguments are the checkbox name and values, respectively (−name and −values). As in the popup menu, the second argument should be an array reference. These values are used for the user-readable labels printed next to the checkboxes as well as for the values passed to your script in the query string.
2. The optional third argument (−default) can be either a reference to a list containing the values to be checked by default, or can be a single value to checked. If this argument is missing or undefined, then nothing is selected when the list first appears.
3. The optional fourth argument (−linebreak) can be set to true to place line breaks between the checkboxes so that they appear as a vertical list. Otherwise, they will be strung together on a horizontal line.
4. The optional fifth argument is a pointer to an associative array relating the checkbox values to the user-visible labels that will will be printed next to them (−labels). If not provided, the values will be used as the default.
5. **HTML3-compatible browsers** (such as Netscape) can take advantage of the optional parameters **−rows**, and **−columns**. These parameters cause *checkbox_group()* to return an HTML3 compatible table containing the checkbox group formatted with the specified number of rows and columns. You can provide just the −columns parameter if you wish; checkbox_group will calculate the correct number of rows for you.

   To include row and column headings in the returned table, you can use the **−rowheader** and **−colheader** parameters. Both of these accept a pointer to an array of headings to use. The headings are just decorative. They don't reorganize the interpretation of the checkboxes -- they're still a single named unit.

When the form is processed, all checked boxes will be returned as a list under the parameter name 'group_name'. The values of the ''on'' checkboxes can be retrieved with:

```
@turned_on = $query->param('group_name');
```

The value returned by *checkbox_group()* is actually an array of button elements. You can capture them and use them within tables, lists, or in other creative ways:

```
@h = $query->checkbox_group(-name=>'group_name',-values=>\@values);
&use_in_creative_way(@h);
```

JAVASCRIPTING: *checkbox_group()* recognizes the **−onClick** parameter. This specifies a JavaScript code fragment or function call to be executed every time the user clicks on any of the buttons in the group. You can retrieve the identity of the particular button clicked on using the ''this'' variable.

CREATING A STANDALONE CHECKBOX

```
print $query->checkbox(-name=>'checkbox_name',
                       -checked=>'checked',
                       -value=>'ON',
                       -label=>'CLICK ME');

    -or-
```

```
print $query->checkbox('checkbox_name','checked','ON','CLICK ME');
```

checkbox() is used to create an isolated checkbox that isn't logically related to any others.

Parameters:

1. The first parameter is the required name for the checkbox (−name). It will also be used for the user-readable label printed next to the checkbox.
2. The optional second parameter (−checked) specifies that the checkbox is turned on by default. Synonyms are −selected and −on.
3. The optional third parameter (−value) specifies the value of the checkbox when it is checked. If not provided, the word "on" is assumed.
4. The optional fourth parameter (−label) is the user-readable label to be attached to the checkbox. If not provided, the checkbox name is used.

The value of the checkbox can be retrieved using:

```
$turned_on = $query->param('checkbox_name');
```

JAVASCRIPTING: *checkbox()* recognizes the **−onClick** parameter. See *checkbox_group()* for further details.

CREATING A RADIO BUTTON GROUP

```
print $query->radio_group(-name=>'group_name',
                          -values=>['eenie','meenie','minie'],
                          -default=>'meenie',
                          -linebreak=>'true',
                          -labels=>\%labels);

     -or-

print $query->radio_group('group_name',['eenie','meenie','minie'],
                                       'meenie','true',\%labels);

HTML3-COMPATIBLE BROWSERS ONLY:

print $query->radio_group(-name=>'group_name',
                          -values=>['eenie','meenie','minie','moe'],
                          -rows=2,-columns=>2);
```

radio_group() creates a set of logically-related radio buttons (turning one member of the group on turns the others off)

Parameters:

1. The first argument is the name of the group and is required (−name).
2. The second argument (−values) is the list of values for the radio buttons. The values and the labels that appear on the page are identical. Pass an array *reference* in the second argument, either using an anonymous array, as shown, or by referencing a named array as in "\@foo".
3. The optional third parameter (−default) is the name of the default button to turn on. If not specified, the first item will be the default. You can provide a nonexistent button name, such as "−" to start up with no buttons selected.
4. The optional fourth parameter (−linebreak) can be set to 'true' to put line breaks between the buttons, creating a vertical list.

5. The optional fifth parameter (−labels) is a pointer to an associative array relating the radio button values to user-visible labels to be used in the display. If not provided, the values themselves are displayed.

6. **HTML3-compatible browsers** (such as Netscape) can take advantage of the optional parameters **−rows**, and **−columns**. These parameters cause *radio_group()* to return an HTML3 compatible table containing the radio group formatted with the specified number of rows and columns. You can provide just the −columns parameter if you wish; radio_group will calculate the correct number of rows for you.

   To include row and column headings in the returned table, you can use the **−rowheader** and **−colheader** parameters. Both of these accept a pointer to an array of headings to use. The headings are just decorative. They don't reorganize the interpetation of the radio buttons -- they're still a single named unit.

When the form is processed, the selected radio button can be retrieved using:

```
      $which_radio_button = $query->param('group_name');
```

The value returned by *radio_group()* is actually an array of button elements. You can capture them and use them within tables, lists, or in other creative ways:

```
    @h = $query->radio_group(-name=>'group_name',-values=>\@values);
    &use_in_creative_way(@h);
```

CREATING A SUBMIT BUTTON

```
   print $query->submit(-name=>'button_name',
                        -value=>'value');

       -or-

   print $query->submit('button_name','value');
```

submit() will create the query submission button. Every form should have one of these.

Parameters:

1. The first argument (−name) is optional. You can give the button a name if you have several submission buttons in your form and you want to distinguish between them. The name will also be used as the user-visible label. Be aware that a few older browsers don't deal with this correctly and **never** send back a value from a button.

2. The second argument (−value) is also optional. This gives the button a value that will be passed to your script in the query string.

You can figure out which button was pressed by using different values for each one:

```
     $which_one = $query->param('button_name');
```

JAVASCRIPTING: *radio_group()* recognizes the **−onClick** parameter. See *checkbox_group()* for further details.

CREATING A RESET BUTTON

```
   print $query->reset
```

reset() creates the "reset" button. Note that it restores the form to its value from the last time the script was called, NOT necessarily to the defaults.

CREATING A DEFAULT BUTTON

```
print $query->defaults('button_label')
```

defaults() creates a button that, when invoked, will cause the form to be completely reset to its defaults, wiping out all the changes the user ever made.

CREATING A HIDDEN FIELD

```
        print $query->hidden(-name=>'hidden_name',
                             -default=>['value1','value2'...]);

                -or-

        print $query->hidden('hidden_name','value1','value2'...);
```

hidden() produces a text field that can't be seen by the user. It is useful for passing state variable information from one invocation of the script to the next.

Parameters:

1. The first argument is required and specifies the name of this field (–name).
2. The second argument is also required and specifies its value (–default). In the named parameter style of calling, you can provide a single value here or a reference to a whole list

Fetch the value of a hidden field this way:

```
$hidden_value = $query->param('hidden_name');
```

Note, that just like all the other form elements, the value of a hidden field is "sticky". If you want to replace a hidden field with some other values after the script has been called once you'll have to do it manually:

```
$query->param('hidden_name','new','values','here');
```

CREATING A CLICKABLE IMAGE BUTTON

```
print $query->image_button(-name=>'button_name',
                           -src=>'/source/URL',
                           -align=>'MIDDLE');

   -or-

print $query->image_button('button_name','/source/URL','MIDDLE');
```

image_button() produces a clickable image. When it's clicked on the position of the click is returned to your script as "button_name.x" and "button_name.y", where "button_name" is the name you've assigned to it.

JAVASCRIPTING: *image_button()* recognizes the **–onClick** parameter. See *checkbox_group()* for further details.

Parameters:

1. The first argument (–name) is required and specifies the name of this field.
2. The second argument (–src) is also required and specifies the URL

3. The third option (−align, optional) is an alignment type, and may be TOP, BOTTOM or MIDDLE

Fetch the value of the button this way:

```
$x = $query->param('button_name.x');
$y = $query->param('button_name.y');
```

CREATING A JAVASCRIPT ACTION BUTTON

```
print $query->button(-name=>'button_name',
                     -value=>'user visible label',
                     -onClick=>"do_something()");

   -or-

print $query->button('button_name',"do_something()");
```

button() produces a button that is compatible with Netscape 2.0's JavaScript. When it's pressed the fragment of JavaScript code pointed to by the **−onClick** parameter will be executed. On non-Netscape browsers this form element will probably not even display.

NETSCAPE COOKIES

Netscape browsers versions 1.1 and higher support a so-called "cookie" designed to help maintain state within a browser session. CGI.pm has several methods that support cookies.

A cookie is a name=value pair much like the named parameters in a CGI query string. CGI scripts create one or more cookies and send them to the browser in the HTTP header. The browser maintains a list of cookies that belong to a particular Web server, and returns them to the CGI script during subsequent interactions.

In addition to the required name=value pair, each cookie has several optional attributes:

1. an expiration time

   This is a time/date string (in a special GMT format) that indicates when a cookie expires. The cookie will be saved and returned to your script until this expiration date is reached if the user exits Netscape and restarts it. If an expiration date isn't specified, the cookie will remain active until the user quits Netscape.

2. a domain

   This is a partial or complete domain name for which the cookie is valid. The browser will return the cookie to any host that matches the partial domain name. For example, if you specify a domain name of ".capricorn.com", then Netscape will return the cookie to Web servers running on any of the machines "www.capricorn.com", "www2.capricorn.com", "feckless.capricorn.com", etc. Domain names must contain at least two periods to prevent attempts to match on top level domains like ".edu". If no domain is specified, then the browser will only return the cookie to servers on the host the cookie originated from.

3. a path

   If you provide a cookie path attribute, the browser will check it against your script's URL before returning the cookie. For example, if you specify the path "/cgi-bin", then the cookie will be returned to each of the scripts "/cgi-bin/tally.pl", "/cgi-bin/order.pl", and "/cgi-bin/customer_service/complain.pl", but not to the script "/cgi-private/site_admin.pl". By default, path is set to "/", which causes the cookie to be sent to any CGI script on your site.

4. a 'secure' flag

   If the "secure" attribute is set, the cookie will only be sent to your script if the CGI request is occurring on a secure channel, such as SSL.

The interface to Netscape cookies is the **cookie()** method:

```
$cookie = $query->cookie(-name=>'sessionID',
                         -value=>'xyzzy',
                         -expires=>'+1h',
                         -path=>'/cgi-bin/database',
                         -domain=>'.capricorn.org',
                         -secure=>1);
print $query->header(-cookie=>$cookie);
```

cookie() creates a new cookie. Its parameters include:

–name
The name of the cookie (required). This can be any string at all. Although Netscape limits its cookie names to non-whitespace alphanumeric characters, CGI.pm removes this restriction by escaping and unescaping cookies behind the scenes.

–value
The value of the cookie. This can be any scalar value, array reference, or even associative array reference. For example, you can store an entire associative array into a cookie this way:

```
$cookie=$query->cookie(-name=>'family information',
                       -value=>\%childrens_ages);
```

–path
The optional partial path for which this cookie will be valid, as described above.

–domain
The optional partial domain for which this cookie will be valid, as described above.

–expires
The optional expiration date for this cookie. The format is as described in the section on the **header()** method:

```
"+1h"  one hour from now
```

–secure
If set to true, this cookie will only be used within a secure SSL session.

The cookie created by *cookie()* must be incorporated into the HTTP header within the string returned by the *header()* method:

```
print $query->header(-cookie=>$my_cookie);
```

To create multiple cookies, give *header()* an array reference:

```
$cookie1 = $query->cookie(-name=>'riddle_name',
                          -value=>"The Sphynx's Question");
$cookie2 = $query->cookie(-name=>'answers',
                          -value=>\%answers);
print $query->header(-cookie=>[$cookie1,$cookie2]);
```

To retrieve a cookie, request it by name by calling *cookie()* method without the **–value** parameter:

```
use CGI;
$query = new CGI;
%answers = $query->cookie(-name=>'answers');
# $query->cookie('answers') will work too!
```

The cookie and CGI namespaces are separate. If you have a parameter named 'answers' and a cookie named 'answers', the values retrieved by *param()* and *cookie()* are independent of each other. However, it's simple to turn a CGI parameter into a cookie, and vice-versa:

```
# turn a CGI parameter into a cookie
$c=$q->cookie(-name=>'answers',-value=>[$q->param('answers')]);
# vice-versa
$q->param(-name=>'answers',-value=>[$q->cookie('answers')]);
```

See the **cookie.cgi** example script for some ideas on how to use cookies effectively.

NOTE: There appear to be some (undocumented) restrictions on Netscape cookies. In Netscape 2.01, at least, I haven't been able to set more than three cookies at a time. There may also be limits on the length of cookies. If you need to store a lot of information, it's probably better to create a unique session ID, store it in a cookie, and use the session ID to locate an external file/database saved on the server's side of the connection.

WORKING WITH NETSCAPE FRAMES

It's possible for CGI.pm scripts to write into several browser panels and windows using Netscape's frame mechanism. There are three techniques for defining new frames programmatically:

1. Create a <Frameset> document

   After writing out the HTTP header, instead of creating a standard HTML document using the *start_html()* call, create a <FRAMESET> document that defines the frames on the page. Specify your *script*(s) (with appropriate parameters) as the SRC for each of the frames.

   There is no specific support for creating <FRAMESET> sections in CGI.pm, but the HTML is very simple to write. See the frame documentation in Netscape's home pages for details

   ```
   http://home.netscape.com/assist/net_sites/frames.html
   ```

2. Specify the destination for the document in the HTTP header

   You may provide a **–target** parameter to the *header()* method:

   print $q–>*header*(–target=>'ResultsWindow');

   This will tell Netscape to load the output of your script into the frame named "ResultsWindow". If a frame of that name doesn't already exist, Netscape will pop up a new window and load your script's document into that. There are a number of magic names that you can use for targets. See the frame documents on Netscape's home pages for details.

3. Specify the destination for the document in the <FORM> tag

   You can specify the frame to load in the FORM tag itself. With CGI.pm it looks like this:

   ```
   print $q->startform(-target=>'ResultsWindow');
   ```

   When your script is reinvoked by the form, its output will be loaded into the frame named "ResultsWindow". If one doesn't already exist a new window will be created.

The script "frameset.cgi" in the examples directory shows one way to create pages in which the fill-out form and the response live in side-by-side frames.

DEBUGGING

If you are running the script from the command line or in the perl debugger, you can pass the script a list of keywords or parameter=value pairs on the command line or from standard input (you don't have to worry about tricking your script into reading from environment variables). You can pass keywords like this:

```
your_script.pl keyword1 keyword2 keyword3
```

or this:

```
your_script.pl keyword1+keyword2+keyword3
```

or this:

```
your_script.pl name1=value1 name2=value2
```

or this:

```
your_script.pl name1=value1&name2=value2
```

or even as newline-delimited parameters on standard input.

When debugging, you can use quotes and backslashes to escape characters in the familiar shell manner, letting you place spaces and other funny characters in your parameter=value pairs:

```
your_script.pl "name1='I am a long value'" "name2=two\ words"
```

DUMPING OUT ALL THE NAME/VALUE PAIRS

The *dump()* method produces a string consisting of all the query's name/value pairs formatted nicely as a nested list. This is useful for debugging purposes:

```
print $query->dump
```

Produces something that looks like:

```
<UL>
<LI>name1
    <UL>
    <LI>value1
    <LI>value2
    </UL>
<LI>name2
    <UL>
    <LI>value1
    </UL>
</UL>
```

You can pass a value of 'true' to *dump()* in order to get it to print the results out as plain text, suitable for incorporating into a <PRE> section.

As a shortcut, as of version 1.56 you can interpolate the entire CGI object into a string and it will be replaced with the the a nice HTML dump shown above:

```
$query=new CGI;
print "<H2>Current Values</H2> $query\n";
```

FETCHING ENVIRONMENT VARIABLES

Some of the more useful environment variables can be fetched through this interface. The methods are as follows:

accept()

Return a list of MIME types that the remote browser accepts. If you give this method a single argument corresponding to a MIME type, as in `$query->`*accept*('text/html'), it will return a floating point value corresponding to the browser's preference for this type from 0.0 (don't want) to 1.0. Glob types (e.g. text/*) in the browser's accept list are handled correctly.

raw_cookie()

Returns the HTTP_COOKIE variable, an HTTP extension implemented by Netscape browsers version 1.1 and higher. Cookies have a special format, and this method call just returns the raw form (?cookie dough). See *cookie()* for ways of setting and retrieving cooked cookies.

user_agent()
Returns the HTTP_USER_AGENT variable. If you give this method a single argument, it will attempt to pattern match on it, allowing you to do something like `$query->`*user_agent*(netscape);

path_info()
Returns additional path information from the script URL. E.G. fetching /cgi-bin/your_script/additional/stuff will result in `$query->`*path_info()* returning "additional/stuff".

NOTE: The Microsoft Internet Information Server is broken with respect to additional path information. If you use the Perl DLL library, the IIS server will attempt to execute the additional path information as a Perl script. If you use the ordinary file associations mapping, the path information will be present in the environment, but incorrect. The best thing to do is to avoid using additional path information in CGI scripts destined for use with IIS.

path_translated()
As per *path_info()* but returns the additional path information translated into a physical path, e.g. "/usr/local/etc/httpd/htdocs/additional/stuff".

The Microsoft IIS is broken with respect to the translated path as well.

remote_host()
Returns either the remote host name or IP address. if the former is unavailable.

script_name() Return the script name as a partial URL, for self-refering scripts.

referer()
Return the URL of the page the browser was viewing prior to fetching your script. Not available for all browsers.

auth_type ()
Return the authorization/verification method in use for this script, if any.

server_name ()
Returns the name of the server, usually the machine's host name.

virtual_host ()
When using virtual hosts, returns the name of the host that the browser attempted to contact

server_software ()
Returns the server software and version number.

remote_user ()
Return the authorization/verification name used for user verification, if this script is protected.

user_name ()
Attempt to obtain the remote user's name, using a variety of different techniques. This only works with older browsers such as Mosaic. Netscape does not reliably report the user name!

request_method()
Returns the method used to access your script, usually one of 'POST', 'GET' or 'HEAD'.

CREATING HTML ELEMENTS:

In addition to its shortcuts for creating form elements, CGI.pm defines general HTML shortcut methods as well. HTML shortcuts are named after a single HTML element and return a fragment of HTML text that you can then print or manipulate as you like.

This example shows how to use the HTML methods:

```
$q = new CGI;
print $q->blockquote(
                     "Many years ago on the island of",
                     $q->a({href=>"http://crete.org/"},"Crete"),
                     "there lived a minotaur named",
                     $q->strong("Fred."),
                    ),
       $q->hr;
```

This results in the following HTML code (extra newlines have been added for readability):

```
<blockquote>
Many years ago on the island of
<a HREF="http://crete.org/">Crete</a> there lived
a minotaur named <strong>Fred.</strong>
</blockquote>
<hr>
```

If you find the syntax for calling the HTML shortcuts awkward, you can import them into your namespace and dispense with the object syntax completely (see the next section for more details):

```
use CGI shortcuts;      # IMPORT HTML SHORTCUTS
print blockquote(
             "Many years ago on the island of",
             a({href=>"http://crete.org/"},"Crete"),
             "there lived a minotaur named",
             strong("Fred."),
             ),
       hr;
```

PROVIDING ARGUMENTS TO HTML SHORTCUTS

The HTML methods will accept zero, one or multiple arguments. If you provide no arguments, you get a single tag:

```
print hr;
#  gives "<hr>"
```

If you provide one or more string arguments, they are concatenated together with spaces and placed between opening and closing tags:

```
print h1("Chapter","1");
# gives "<h1>Chapter 1</h1>"
```

If the first argument is an associative array reference, then the keys and values of the associative array become the HTML tag's attributes:

```
print a({href=>'fred.html',target=>'_new'},
        "Open a new frame");
# gives <a href="fred.html",target="_new">Open a new frame</a>
```

You are free to use CGI.pm-style dashes in front of the attribute names if you prefer:

```
print img {-src=>'fred.gif',-align=>'LEFT'};
# gives <img ALIGN="LEFT" SRC="fred.gif">
```

Generating new HTML tags

Since no mere mortal can keep up with Netscape and Microsoft as they battle it out for control of HTML, the code that generates HTML tags is general and extensible. You can create new HTML tags freely just by referring to them on the import line:

```
use CGI shortcuts,winkin,blinkin,nod;
```

Now, in addition to the standard CGI shortcuts, you've created HTML tags named "winkin", "blinkin" and "nod". You can use them like this:

```
print blinkin {color=>'blue',rate=>'fast'},"Yahoo!";
# <blinkin COLOR="blue" RATE="fast">Yahoo!</blinkin>
```

IMPORTING CGI METHOD CALLS INTO YOUR NAME SPACE

As a convenience, you can import most of the CGI method calls directly into your name space. The syntax for doing this is:

```
use CGI <list of methods>;
```

The listed methods will be imported into the current package; you can call them directly without creating a CGI object first. This example shows how to import the **param()** and **header()** methods, and then use them directly:

```
use CGI param,header;
print header('text/plain');
$zipcode = param('zipcode');
```

You can import groups of methods by referring to a number of special names:

cgi Import all CGI–handling methods, such as **param()**, **path_info()** and the like.

form
Import all fill-out form generating methods, such as **textfield()**.

html2
Import all methods that generate HTML 2.0 standard elements.

html3
Import all methods that generate HTML 3.0 proposed elements (such as <table>, <super> and <sub>).

netscape
Import all methods that generate Netscape-specific HTML extensions.

shortcuts
Import all HTML–generating shortcuts (i.e. 'html2' + 'html3' + 'netscape')...

standard
Import "standard" features, 'html2', 'form' and 'cgi'.

all Import all the available methods. For the full list, see the CGI.pm code, where the variable `%TAGS` is defined.

Note that in the interests of execution speed CGI.pm does **not** use the standard the *Exporter* manpage syntax for specifying load symbols. This may change in the future.

If you import any of the state-maintaining CGI or form-generating methods, a default CGI object will be created and initialized automatically the first time you use any of the methods that require one to be present. This includes **param()**, **textfield()**, **submit()** and the like. (If you need direct access to the CGI object, you can find it in the global variable **$CGI::Q**). By importing CGI.pm methods, you can create visually elegant scripts:

```
use CGI standard,html2;
print
    header,
    start_html('Simple Script'),
    h1('Simple Script'),
    start_form,
    "What's your name? ",textfield('name'),p,
    "What's the combination?",
    checkbox_group(-name=>'words',
                   -values=>['eenie','meenie','minie','moe'],
                   -defaults=>['eenie','moe']),p,
    "What's your favorite color?",
    popup_menu(-name=>'color',
               -values=>['red','green','blue','chartreuse']),p,
    submit,
    end_form,
    hr,"\n";

 if (param) {
    print
        "Your name is ",em(param('name')),p,
        "The keywords are: ",em(join(", ",param('words'))),p,
        "Your favorite color is ",em(param('color')),".\n";
 }
 print end_html;
```

USING NPH SCRIPTS

NPH, or "no-parsed-header", scripts bypass the server completely by sending the complete HTTP header directly to the browser. This has slight performance benefits, but is of most use for taking advantage of HTTP extensions that are not directly supported by your server, such as server push and PICS headers.

Servers use a variety of conventions for designating CGI scripts as NPH. Many Unix servers look at the beginning of the script's name for the prefix "nph-". The Macintosh WebSTAR server and Microsoft's Internet Information Server, in contrast, try to decide whether a program is an NPH script by examining the first line of script output.

CGI.pm supports NPH scripts with a special NPH mode. When in this mode, CGI.pm will output the necessary extra header information when the *header()* and *redirect()* methods are called.

The Microsoft Internet Information Server requires NPH mode. As of version 2.30, CGI.pm will automatically detect when the script is running under IIS and put itself into this mode. You do not need to do this manually, although it won't hurt anything if you do.

There are a number of ways to put CGI.pm into NPH mode:

In the **use** statement Simply add ':nph' to the list of symbols to be imported into your script:

```
use CGI qw(:standard :nph)
```

By calling the **nph()** method:

Call **nph()** with a non-zero parameter at any point after using CGI.pm in your program.

```
CGI->nph(1)
```

By using **–nph** parameters in the **header()** and **redirect()** statements:

```
print $q->header(-nph=&gt;1);
```

AUTHOR INFORMATION

Address bug reports and comments to: lstein@genome.wi.mit.edu

CREDITS

Thanks very much to:

Matt Heffron (heffron@falstaff.css.beckman.com)

James Taylor (james.taylor@srs.gov)

Scott Anguish <sanguish@digifix.com>

Mike Jewell (mlj3u@virginia.edu)

Timothy Shimmin (tes@kbs.citri.edu.au)

Joergen Haegg (jh@axis.se)

Laurent Delfosse (delfosse@csgrad1.cs.wvu.edu)

Richard Resnick (applepi1@aol.com)

Craig Bishop (csb@barwonwater.vic.gov.au)

Tony Curtis (tc@vcpc.univie.ac.at)

Tim Bunce (Tim.Bunce@ig.co.uk)

Tom Christiansen (tchrist@convex.com)

Andreas Koenig (k@franz.ww.TU−Berlin.DE)

Tim MacKenzie (Tim.MacKenzie@fulcrum.com.au)

Kevin B. Hendricks (kbhend@dogwood.tyler.wm.edu)

Stephen Dahmen (joyfire@inxpress.net)

Ed Jordan (ed@fidalgo.net)

David Alan Pisoni (david@cnation.com)

...and many many more...
for suggestions and bug fixes.

A COMPLETE EXAMPLE OF A SIMPLE FORM−BASED SCRIPT

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

use CGI;

$query = new CGI;
```

```
print $query->header;
print $query->start_html("Example CGI.pm Form");
print "<H1> Example CGI.pm Form</H1>\n";
&print_prompt($query);
&do_work($query);
&print_tail;
print $query->end_html;

sub print_prompt {
   my($query) = @_;

   print $query->startform;
   print "<EM>What's your name?</EM><BR>";
   print $query->textfield('name');
   print $query->checkbox('Not my real name');

   print "<P><EM>Where can you find English Sparrows?</EM><BR>";
   print $query->checkbox_group(
                         -name=>'Sparrow locations',
                         -values=>[England,France,Spain,Asia,Hoboken],
                         -linebreak=>'yes',
                         -defaults=>[England,Asia]);

   print "<P><EM>How far can they fly?</EM><BR>",
        $query->radio_group(
                -name=>'how far',
                -values=>['10 ft','1 mile','10 miles','real far'],
                -default=>'1 mile');

   print "<P><EM>What's your favorite color?</EM>  ";
   print $query->popup_menu(-name=>'Color',
                            -values=>['black','brown','red','yellow'],
                            -default=>'red');

   print $query->hidden('Reference','Monty Python and the Holy Grail');

   print "<P><EM>What have you got there?</EM><BR>";
   print $query->scrolling_list(
                 -name=>'possessions',
                 -values=>['A Coconut','A Grail','An Icon',
                           'A Sword','A Ticket'],
                 -size=>5,
                 -multiple=>'true');

   print "<P><EM>Any parting comments?</EM><BR>";
   print $query->textarea(-name=>'Comments',
                          -rows=>10,
                          -columns=>50);

   print "<P>",$query->reset;
   print $query->submit('Action','Shout');
   print $query->submit('Action','Scream');
   print $query->endform;
   print "<HR>\n";
```

```
}

sub do_work {
   my($query) = @_;
   my(@values,$key);

   print "<H2>Here are the current settings in this form</H2>";

   foreach $key ($query->param) {
      print "<STRONG>$key</STRONG> -> ";
      @values = $query->param($key);
      print join(", ",@values),"<BR>\n";
  }
}

sub print_tail {
   print <<END;
<HR>
<ADDRESS>Lincoln D. Stein</ADDRESS><BR>
<A HREF="/">Home Page</A>
END
}
```

BUGS

This module has grown large and monolithic. Furthermore it's doing many things, such as handling URLs, parsing CGI input, writing HTML, etc., that are also done in the LWP modules. It should be discarded in favor of the CGI::* modules, but somehow I continue to work on it.

Note that the code is truly contorted in order to avoid spurious warnings when programs are run with the **−w** switch.

SEE ALSO

the *CGI::Carp* manpage, the *URI::URL* manpage, the *CGI::Request* manpage, the *CGI::MiniSvr* manpage, the *CGI::Base* manpage, the *CGI::Form* manpage, the *CGI::Apache* manpage, the *CGI::Switch* manpage, the *CGI::Push* manpage, the *CGI::Fast* manpage

NAME

CGI::Apache – Make things work with CGI.pm against Perl-Apache API

SYNOPSIS

```
require CGI::Apache;

my $q = new Apache::CGI;

$q->print($q->header);

#do things just like you do with CGI.pm
```

DESCRIPTION

When using the Perl-Apache API, your applications are faster, but the enviroment is different than CGI. This module attempts to set-up that environment as best it can.

NOTE

This module used to be named Apache::CGI. Sorry for the confusion.

SEE ALSO

perl(1), *Apache*(3), *CGI*(3)

AUTHOR

Doug MacEachern <dougm@osf.org>, hacked over by Andreas König <a.koenig@mind.de>, modified by Lincoln Stein <lt>lstein@genome.wi.mit.edu<gt>

NAME

CGI::Carp – CGI routines for writing to the HTTPD (or other) error log

SYNOPSIS

```
use CGI::Carp;

croak "We're outta here!";
confess "It was my fault: $!";
carp "It was your fault!";
warn "I'm confused";
die  "I'm dying.\n";
```

DESCRIPTION

CGI scripts have a nasty habit of leaving warning messages in the error logs that are neither time stamped nor fully identified. Tracking down the script that caused the error is a pain. This fixes that. Replace the usual

```
use Carp;
```

with

```
use CGI::Carp
```

And the standard *warn()*, die (), *croak()*, *confess()* and *carp()* calls will automagically be replaced with functions that write out nicely time-stamped messages to the HTTP server error log.

For example:

```
[Fri Nov 17 21:40:43 1995] test.pl: I'm confused at test.pl line 3.
[Fri Nov 17 21:40:43 1995] test.pl: Got an error message: Permission denied.
[Fri Nov 17 21:40:43 1995] test.pl: I'm dying.
```

REDIRECTING ERROR MESSAGES

By default, error messages are sent to STDERR. Most HTTPD servers direct STDERR to the server's error log. Some applications may wish to keep private error logs, distinct from the server's error log, or they may wish to direct error messages to STDOUT so that the browser will receive them.

The `carpout()` function is provided for this purpose. Since *carpout()* is not exported by default, you must import it explicitly by saying

```
use CGI::Carp qw(carpout);
```

The *carpout()* function requires one argument, which should be a reference to an open filehandle for writing errors. It should be called in a `BEGIN` block at the top of the CGI application so that compiler errors will be caught. Example:

```
BEGIN {
  use CGI::Carp qw(carpout);
  open(LOG, ">>/usr/local/cgi-logs/mycgi-log") or
    die("Unable to open mycgi-log: $!\n");
  carpout(LOG);
}
```

carpout() does not handle file locking on the log for you at this point.

The real STDERR is not closed -- it is moved to SAVEERR. Some servers, when dealing with CGI scripts, close their connection to the browser when the script closes STDOUT and STDERR. SAVEERR is used to prevent this from happening prematurely.

You can pass filehandles to *carpout()* in a variety of ways. The "correct" way according to Tom

Christiansen is to pass a reference to a filehandle GLOB:

```
carpout(\*LOG);
```

This looks weird to mere mortals however, so the following syntaxes are accepted as well:

```
carpout(LOG);
carpout(main::LOG);
carpout(main'LOG);
carpout(\LOG);
carpout(\'main::LOG');

... and so on
```

Use of *carpout()* is not great for performance, so it is recommended for debugging purposes or for moderate-use applications. A future version of this module may delay redirecting STDERR until one of the CGI::Carp methods is called to prevent the performance hit.

MAKING PERL ERRORS APPEAR IN THE BROWSER WINDOW

If you want to send fatal (die, confess) errors to the browser, ask to import the special ‘‘fatalsToBrowser’’ subroutine:

```
use CGI::Carp qw(fatalsToBrowser);
die "Bad error here";
```

Fatal errors will now be echoed to the browser as well as to the log. CGI::Carp arranges to send a minimal HTTP header to the browser so that even errors that occur in the early compile phase will be seen. Nonfatal errors will still be directed to the log file only (unless redirected with carpout).

CHANGE LOG

1.05 *carpout()* added and minor corrections by Marc Hedlund
<hedlund@best.com> on 11/26/95.

1.06 *fatalsToBrowser()* no longer aborts for fatal errors within
eval() statements.

AUTHORS

Lincoln D. Stein <lstein@genome.wi.mit.edu>. Feel free to redistribute this under the Perl Artistic License.

SEE ALSO

Carp, CGI::Base, CGI::BasePlus, CGI::Request, CGI::MiniSvr, CGI::Form, CGI::Response

NAME

CGI::Fast − CGI Interface for Fast CGI

SYNOPSIS

```
use CGI::Fast qw(:standard);
$COUNTER = 0;
while (new CGI::Fast) {
    print header;
    print start_html("Fast CGI Rocks");
    print
        h1("Fast CGI Rocks"),
        "Invocation number ",b($COUNTER++),
        " PID ",b($$),".",
        hr;
    print end_html;
}
```

DESCRIPTION

CGI::Fast is a subclass of the CGI object created by CGI.pm. It is specialized to work well with the Open Market FastCGI standard, which greatly speeds up CGI scripts by turning them into persistently running server processes. Scripts that perform time-consuming initialization processes, such as loading large modules or opening persistent database connections, will see large performance improvements.

OTHER PIECES OF THE PUZZLE

In order to use CGI::Fast you'll need a FastCGI−enabled Web server. Open Market's server is FastCGI−savvy. There are also freely redistributable FastCGI modules for NCSA httpd 1.5 and Apache. FastCGI−enabling modules for Microsoft Internet Information Server and Netscape Communications Server have been announced.

In addition, you'll need a version of the Perl interpreter that has been linked with the FastCGI I/O library. Precompiled binaries are available for several platforms, including DEC Alpha, HP−UX and SPARC/Solaris, or you can rebuild Perl from source with patches provided in the FastCGI developer's kit. The FastCGI Perl interpreter can be used in place of your normal Perl without ill consequences.

You can find FastCGI modules for Apache and NCSA httpd, precompiled Perl interpreters, and the FastCGI developer's kit all at URL:

```
http://www.fastcgi.com/
```

WRITING FASTCGI PERL SCRIPTS

FastCGI scripts are persistent: one or more copies of the script are started up when the server initializes, and stay around until the server exits or they die a natural death. After performing whatever one-time initialization it needs, the script enters a loop waiting for incoming connections, processing the request, and waiting some more.

A typical FastCGI script will look like this:

```
#!/usr/local/bin/perl    # must be a FastCGI version of perl!
use CGI::Fast;
&do_some_initialization();
while ($q = new CGI::Fast) {
    &process_request($q);
}
```

Each time there's a new request, CGI::Fast returns a CGI object to your loop. The rest of the time your script waits in the call to *new()*. When the server requests that your script be terminated, *new()* will return undef. You can of course exit earlier if you choose. A new version of the script will be respawned to take its place (this may be necessary in order to avoid Perl memory leaks in long-running scripts).

CGI.pm's default CGI object mode also works. Just modify the loop this way:

```
while (new CGI::Fast) {
    &process_request;
}
```

Calls to *header()*, *start_form()*, etc. will all operate on the current request.

INSTALLING FASTCGI SCRIPTS

See the FastCGI developer's kit documentation for full details. On the Apache server, the following line must be added to srm.conf:

```
AddType application/x-httpd-fcgi .fcgi
```

FastCGI scripts must end in the extension .fcgi. For each script you install, you must add something like the following to srm.conf:

```
AppClass /usr/etc/httpd/fcgi-bin/file_upload.fcgi -processes 2
```

This instructs Apache to launch two copies of file_upload.fcgi at startup time.

USING FASTCGI SCRIPTS AS CGI SCRIPTS

Any script that works correctly as a FastCGI script will also work correctly when installed as a vanilla CGI script. However it will not see any performance benefit.

CAVEATS

I haven't tested this very much.

AUTHOR INFORMATION

be used and modified freely, but I do request that this copyright notice remain attached to the file. You may modify this module as you wish, but if you redistribute a modified version, please attach a note listing the modifications you have made.

Address bug reports and comments to: lstein@genome.wi.mit.edu

BUGS

This section intentionally left blank.

SEE ALSO

the *CGI::Carp* manpage, the *CGI* manpage

=cut

NAME

CGI::Push − Simple Interface to Server Push

SYNOPSIS

```
use CGI::Push qw(:standard);

do_push(−next_page=>\&next_page,
        −last_page=>\&last_page,
        −delay=>0.5);

sub next_page {
    my($q,$counter) = @_;
    return undef if $counter >= 10;
    return start_html('Test'),
           h1('Visible'),"\n",
           "This page has been called ", strong($counter)," times",
           end_html();
  }

 sub last_page {
     my($q,$counter) = @_;
     return start_html('Done'),
            h1('Finished'),
            strong($counter),' iterations.',
            end_html;
 }
```

DESCRIPTION

CGI::Push is a subclass of the CGI object created by CGI.pm. It is specialized for server push operations, which allow you to create animated pages whose content changes at regular intervals.

You provide CGI::Push with a pointer to a subroutine that will draw one page. Every time your subroutine is called, it generates a new page. The contents of the page will be transmitted to the browser in such a way that it will replace what was there beforehand. The technique will work with HTML pages as well as with graphics files, allowing you to create animated GIFs.

USING CGI::Push

CGI::Push adds one new method to the standard CGI suite, *do_push()*. When you call this method, you pass it a reference to a subroutine that is responsible for drawing each new page, an interval delay, and an optional subroutine for drawing the last page. Other optional parameters include most of those recognized by the CGI *header()* method.

You may call *do_push()* in the object oriented manner or not, as you prefer:

```
use CGI::Push;
$q = new CGI::Push;
$q->do_push(−next_page=>\&draw_a_page);

    −or−

use CGI::Push qw(:standard);
do_push(−next_page=>\&draw_a_page);
```

Parameters are as follows:

```
do_push(−next_page=>\&my_draw_routine);
```

This required parameter points to a reference to a subroutine responsible for drawing each new page. The subroutine should expect two parameters consisting of the CGI object and a counter indicating the number of times the subroutine has been called. It should return the contents of the page as an **array** of one or more items to print. It can return a false value (or an empty array) in order to abort the redrawing loop and print out the final page (if any)

```
sub my_draw_routine {
    my($q,$counter) = @_;
    return undef if $counter > 100;
    return start_html('testing'),
           h1('testing'),
           "This page called $counter times";
}
```

-last_page
: This optional parameter points to a reference to the subroutine responsible for drawing the last page of the series. It is called after the −next_page routine returns a false value. The subroutine itself should have exactly the same calling conventions as the −next_page routine.

-type
: This optional parameter indicates the content type of each page. It defaults to ‘‘text/html’’. Currently, server push of heterogeneous document types is not supported.

-delay
: This indicates the delay, in seconds, between frames. Smaller delays refresh the page faster. Fractional values are allowed.

If not specified, −delay will default to 1 second

-cookie, −target, −expires
: These have the same meaning as the like-named parameters in *CGI::header()*.

INSTALLING CGI::Push SCRIPTS

Server push scripts **must** be installed as no-parsed-header (NPH) scripts in order to work correctly. On Unix systems, this is most often accomplished by prefixing the script's name with ‘‘nph-’’. Recognition of NPH scripts happens automatically with WebSTAR and Microsoft IIS. Users of other servers should see their documentation for help.

CAVEATS

This is a new module. It hasn't been extensively tested.

AUTHOR INFORMATION

be used and modified freely, but I do request that this copyright notice remain attached to the file. You may modify this module as you wish, but if you redistribute a modified version, please attach a note listing the modifications you have made.

Address bug reports and comments to: lstein@genome.wi.mit.edu

BUGS

This section intentionally left blank.

SEE ALSO

the *CGI::Carp* manpage, the *CGI* manpage

=cut

NAME

CGI::Switch – Try more than one constructors and return the first object available

SYNOPSIS

```
use CGISwitch;

 -or-

use CGI::Switch This, That, CGI::XA, Foo, Bar, CGI;

my $q = new CGI::Switch;
```

DESCRIPTION

Per default the *new()* method tries to call *new()* in the three packages Apache::CGI, CGI::XA, and CGI. It returns the first CGI object it succeeds with.

The import method allows you to set up the default order of the modules to be tested.

SEE ALSO

perl (1), *Apache* (3), *CGI* (3), *CGI::XA* (3)

AUTHOR

Andreas König <a.koenig@mind.de>

NAME

mod_perl − Embed a Perl interpreter in the Apache server

DESCRIPTION

This Apache module embeds a perl interpreter in the HTTP server. One benefit of this is that we are able to run scripts without going through the expensive (fork/exec/parameter passing/parsing, perl-startup time, etc.) CGI interface. Not only will scripts run faster, they have direct access to the C API of the server, allowing the developer to extend Apache server functionality with modules written in Perl.

The current approach of mod_perl is to allocate and construct one perl interpreter when the server starts. At the same time, load, parse and run one perl script, which may pull in other perl code such as your favorite modules. This also allows you to initiate persistent connections such as to a database server. Then, a subroutine in memory is called to handle each request. The interpreter is destroyed upon restart or shutdown of the server.

FAQ

Patrick Kane <modus@enews.com> maintains the mod_perl FAQ available at: http://chaos.dc.enews.com/mod_perl/

Apache/Perl API

See 'perldoc Apache' for info on how to use the Perl-Apache API.

See the lib/ directory for example modules.

See the eg/ directory for example scripts.

mod_perl

For using mod_perl as a CGI replacement, the recommended configuration is as follows:

```
Alias /perl/  /real/path/to/perl-scripts/

<Location /perl>
SetHandler  perl-script
PerlHandler Apache::Registry
Options ExecCGI
</Location>
```

Now, any file accessed under /perl will be handled by mod_perl and the Apache::Registry module. The file must exist and be executable, in addition, 'Options ExecCGI' must be turned on. See the Apache::Registry module for details.

By default, mod_perl does not send any headers by itself, however, you may wish to change this:

```
PerlSendHeader On
```

By default, mod_perl does not setup `%ENV`, you may wish to change this:

```
PerlSetupEnv On
```

With the recommended configuration, these options and Perl linked with sfio (see I/O section), scripts running under Apache::Registry will look just like ''normal'' CGI scripts. See eg/perlio.pl as an example.

You may load additional modules via:

```
PerlModule Apache::SSI SomeOther::Module
```

There is a limit of 10 PerlModule's, if you need more to be loaded when the server starts, use one PerlModule to pull in many or use the PerlScript directive described below.

Optionally:

```
PerlScript  /full/path/to/script_to_load_at_startup.pl
```

This script will be loaded when the server starts. See eg/MyPackage.pl for an example to start with.

In an access.conf <Directory /foo> or .htaccess you need:

```
PerlHandler sub_routine_name
```

This is the name of the subroutine to call to handle each request. e.g. in the PerlScript MyPackage.pl this is "MyPackage::handler". in the PerlModule Apache::Registry this is "Apache::Registry::handler".

There are several stages of a request where the Apache API allows a module to step in and do something. The Apache documentation will tell you all about those stages and what your modules can do. By default, these hooks are disabled at compile time, see the INSTALL document for information on enabling these hooks. The following configuration directives take one argument, which is the name of the subroutine to call. If the value is not a subroutine name, mod_perl assumes it is a class name which implements a 'handler' subroutine.

```
PerlTransHandler
PerlAuthenHandler
PerlAuthzHandler
PerlAccessHandler
PerlTypeHandler
PerlFixupHandler
PerlLogHandler
PerlHeaderParser (requires apache1.2b5 or higher)
```

Two subroutines 'OK' and 'DECLINED' are defined when the server starts to help you configure nested Perl*Handlers. Example, let's say you want to configure a default PerlLogHandler and turn it off for certain sub-directories or files.

```
<Location />
PerlLogHandler MyLogger
....
</Location>

#don't log requests for gif files
<Files *.gif>
PerlLogHandler DECLINED
</Files>
```

I/O

Apache's i/o is not stream oriented. So, by default, you cannot *print()* to STDOUT from your script, use `$r`−>*print()* instead. Nor can you *read()* from STDIN, use `$r`−>read*() or the `$r`−>content methods to read POST data. As of Perl5.003_02, there are two mechanisms in place for redirecting the STDIN and STDOUT streams.

One mechanism takes advantage of the PerlIO abstraction and sfio discipline structures, such that STDIN and STDOUT are hooked up to the client by default if you configured perl with −Dusesfio (see Perl's INSTALL doc).

Otherwise, mod_perl will *tie()* STDOUT to the client. In order for this to work, you must have Perl 5.003_02+ or apply the pp_hot.patch to your Perl source. There is no magic method to read from a tie'd STDIN, so for now you must use `$r`−>read*() or `$r`−>content to fetch client data.

Using CGI.pm and CGI::*

For users of CGI.pm, see the Apache::CGI, CGI::XA and CGI::Switch modules.

The CGI::* modules (CGI::Request etal.) can be used untouched if your Perl is configured to use sfio and the following directives are present in the directory configuration:

```
PerlSendHeader On
PerlSetupEnv   On
```

If you use the `SendHeaders()` function, be sure to call `$req_obj`−>cgi->done when you are done with a request, just as you would under *CGI::MiniSrv*.

MEMORY CONSUMPTION

No matter what, your httpd will be larger than normal to start, simply because you've linked with perl's runtime.

Here's I'm just running

```
% /usr/bin/perl −e ’1 while 1’

  PID USERNAME PRI NICE   SIZE   RES STATE   TIME   WCPU    CPU COMMAND
10214 dougm     67    0   668K  212K run     0:04 71.55% 21.13% perl
```

Now with a few random modules:

```
% /usr/bin/perl −MDBI −MDBD::mSQL −MLWP::UserAgent −MFileHandle −MIO −MPOSIX −e ’1 while 1’

10545 dougm     49    0  3732K 3340K run     0:05 54.59% 21.48% perl
```

Here's my httpd linked with libperl.a, not having served a single request:

```
10386 dougm      5    0  1032K  324K sleep   0:00  0.12%  0.11% httpd−a
```

You can reduce this if you configure perl 5.003_xx+ with −Duseshrplib. Here's my httpd linked with libperl.sl, not having served a single request:

```
10393 dougm      5    0   476K  368K sleep   0:00  0.12%  0.10% httpd−s
```

Now, once the server starts receiving requests, the embedded interpreter will compile code for each ‘require’ file it has not seen yet, each new Apache::Registry subroutine that's compiled, along with whatever modules it's use'ing or require'ing. Not to mention AUTOLOADing. (Modules that you ‘use’ will be compiled when the server starts unless they are inside an eval block.) httpd will grow just as big as our /usr/bin/perl would, or a CGI process for that matter, it all depends on your setup.

For me, once everything is compiled, the processes no longer grow, I can live with the size at that point. For others, this size might be too big, or they might be using a module that leaks or have code of their own that leaks, in any case using the apache configuration directive ‘MaxRequestsPerChild’ is your best bet to keep the size down, but at the same time, you'll be slowing things down when Apache::Registry scripts have to recompile. Tradeoffs...

WARNINGS

Your scripts *cannot* be in a *ScriptAlias* (ed) (which is usually /cgi-bin) directory.

Your scripts *will not* run from the command line (yet) unless you use CGI::Switch and no direct calls to Apache->methods.

SUPPORT

For comments, questions, bug-reports, announcements, etc., send mail to majordomo@listproc.itribe.net with the string “subscribe modperl” in the body.

AUTHOR

Doug MacEachern <dougm@osf.org>

NAME

Apache − Perl interface to the Apache server API

SYNOPSIS

```
use Apache ();
```

DESCRIPTION

This module provides a Perl interface the Apache API. It's here mainly for **mod_perl**, but may be used for other Apache modules that wish to embed a Perl interpreter. We suggest that you also consult the description of the Apache C API at http://www.apache.org/docs/.

THE REQUEST OBJECT

The request object holds all the information that the server needs to service a request. Apache **Perl*Handler**s will be given a reference to the request object as parameter and may choose update or use it in various ways. Most of the methods described below obtain information from or updates the request object. The perl version of the request object will be blessed into the **Apache** package, it is really a `request_rec *` in disguise.

Apache->request([$r])

The Apache->request method will return a reference to the request object.

Perl*Handler's can obtain a reference to the request object when it is passed to them via `@_`. However, scripts that run under the *Apache::Registry* manpage, for example, need a way to access the request object. the *Apache::Registry* manpage will make a request object availible to these scripts by passing it's object reference to `Apache−`*request*($r)>. If handlers use modules such as `Apache::CGI` that need to access the *Apache*-request> manpage, they too should do this (e.g. Apache::Status).

$r->main

If the current request is a sub-request, this method returns a blessed reference to the main request structure.

$r->is_main

Returns true if the current request object is for the main request.

CLIENT REQUEST PARAMETERS

First we will take a look at various methods that can be used to retrieve the request parameters sent from the client. In the following examples, **$r** is a request object blessed into the **Apache** class, obtained by a handler's first parameter or *Apache->request*

$r->method([$meth])

The `$r−>`method method will return the request method. It will be a string such as "GET", "HEAD" or "POST". Passing an argument will set the method, mainly used for internal redirects.

$r->method_number([$num])

The `$r−>`method_number method will return the request method number. Each number corresponds to a string representation such as "GET", "HEAD" or "POST". Passing an argument will set the method_number, mainly used for internal redirects and testing authorization restriction masks.

$r->the_request

The request line send by the client, handy for logging, etc.

$r->proxyreq

Returns true if the request is proxy http. Mainly used during the filename translation stage of the request, which may be handled by a `PerlTransHandler`.

$r->header_only

Returns true if the client is asking for headers only, e.g. if the request method was **HEAD**.

$r->protocol
The $r->protocol method will return a string identifying the protocol that the client speaks. Typical values will be "HTTP/1.0" or "HTTP/1.1".

$r->uri([$uri])
The $r->uri method will return the requested URI, optionally changing it with the first argument.

$r->filename([$filename])
The $r->filename method will return the result of the *URI --> filename* translation, optionally changing it with the first argument if you happen to be doing the translation.

$r->path_info([$path_info])
The $r->path_info method will return what's left in the path after the *URI --> filename* translation, optionally changing it with the first argument if you happen to be doing the translation.

$r->args
The $r->args method will return the contents of the URI's *query string*. When called in a scalar context, the entire string is returned. When called in a list context, a list of parsed *key* => *value* pairs are returned, i.e. it can be used like this:

```
$query = $r->args;
%in    = $r->args;
```

$r->headers_in
The $r->headers_in method will return a %hash of client request headers. This can be used to initialize a perl hash, or one could use the $r->*header_in()* method (described below) to retrieve a specific header value directly.

$r->header_in($header_name, [$value])
Return the value of a client header. Can be used like this:

```
$ct = $r->header_in("Content-type");

$r->header_in($key, $val); #set the value of header '$key'
```

$r->content
The $r->content method will return the entity body read from the client, but only if the request content type is application/x-www-form-urlencoded. When called in a scalar context, the entire string is returned. When called in a list context, a list of parsed *key* => *value* pairs are returned. *NOTE*: you can only ask for this once, as the entire body is read from the client.

$r->read_client_block($buf, $bytes_to_read)
Read from the entity body sent by the client. Example of use:

```
$r->read_client_block($buf, $r->header_in('Content-length'));
```

$r->read($buf, $bytes_to_read)
This method uses *read_client_block()* to read data from the client, looping until it gets all of $bytes_to_read or a timeout happens.

In addition, this method sets a timeout before reading with $r-hard_timeout>

$r->get_remote_host
Lookup the client's DNS hostname. Might return *undef* if the hostname is not known.

More information about the client can be obtained from the **Apache::Connection** object, as described below.

$c = $r->connection
: The `$r->`connection method will return a reference to the request connection object (blessed into the **Apache::Connection** package). This is really a `conn_rec*` in disguise. The following methods can be used on the connection object:

$c->remote_host

$c->remote_ip

$c->remote_logname

$c->user; #Returns the remote username if authenticated.

$c->auth_type; #Returns the authentication scheme used, if any.

$c->aborted; #returns true if the client stopped talking to us

$c->close; #Calling this method will close down the connection to the client

SERVER CONFIGURATION INFORMATION

The following methods are used to obtain information from server configuration and access control files.

$r->dir_config($key)

Returns the value of a per-directory variable specified by the `PerlSetVar` directive.

```
#<Location /foo/bar>
#SetPerlVar  Key  Value
#</Location>

my $val = $r->dir_config('Key');
```

$r->requires

Returns an array reference of hash references, containing information related to the **require** directive. This is normally used for access control, see the *Apache::AuthzAge* manpage for an example.

$r->allow_options

The `$r->`allow_options method can be used for checking if it's ok to run a perl script. The **Apache::Options** module provide the constants to check against.

```
if(!($r->allow_options & OPT_EXECCGI)) {
    $r->log_reason("Options ExecCGI is off in this directory",
                   $filename);
}
```

$s = $r->server

Return a reference to the server info object (blessed into the **Apache::Server** package). This is really a `server_rec*` in disguise. The following methods can be used on the server object:

$s->server_admin

Returns the mail address of the person responsible for this server.

$s->server_hostname

Returns the hostname used by this server.

$s->port

Returns the port that this servers listens too.

$s->is_virtal

Returns true if this is a virtual server.

$s->names
Returns the wildcarded names for HostAlias servers.

SETTING UP THE RESPONSE

The following methods are used to set up and return the response back to the client. This typically involves setting up `$r`−>*status()*, the various content attributes and optionally some additional `$r`−>*header_out()*'s before calling `$r`−>*send_http_header()* which will actually send the headers to the client. After this a typical application will call the `$r`−>*print()* method to send the response content to the client.

$r->send_http_header
Send the response line and all headers to the client. (This method will actually call `$r`−>basic_http_header first).

This method will create headers from the `$r`−>*content_xxx()* and `$r`−>*no_cache()* attributes (described below) and then append the headers defined by `$r`−>header_out (or `$r`−>err_header_out if status indicates an error).

$r->get_basic_auth_pw
If the current request is protected by Basic authentication, this method will return 0, otherwise −1. The second return value will be the decoded password sent by the client.

```
($ret, $sent_pw) = $r->get_basic_auth_pw;
```

$r->note_basic_auth_failure
Prior to requiring Basic authentication from the client, this method will set the outgoing HTTP headers asking the client to authenticate for the realm defined by the configuration directive `AuthName`.

$r->handler([$meth])
Set the handler for a request. Normally set by the configuration directive `AddHandler`.

`$r`−>*handler*("perl-script");

$r->content_type([$newval])
Get or set the content type being sent to the client. Content types are strings like "text/plain", "text/html" or "image/gif". This corresponds to the "Content-Type" header in the HTTP protocol. Example of usage is:

```
$previous_type = $r->content_type;
$r->content_type("text/plain");
```

$r->content_encoding([$newval])
Get or set the content encoding. Content encodings are string like "gzip" or "compress". This correspond to the "Content-Encoding" header in the HTTP protocol.

$r->content_language([$newval])
Get or set the content language. The content language corresponds to the "Content-Language" HTTP header and is a string like "en" or "no".

$r->status($integer)
Get or set the reply status for the client request. The **Apache::Constants** module provide mnemonic names for the status codes.

$r->status_line($string)
Get or set the response status line. The status line is a string like "HTTP/1.0 200 OK" and it will take precedence over the value specified using the `$r`−>*status()* described above.

$r->headers_out
The `$r`−>headers_out method will return a `%hash` of server response headers. This can be used to initialize a perl hash, or one could use the `$r`−>*header_out()* method (described below) to retrieve or set a specific header value directly.

$r->header_out($header, $value)
: Change the value of a response header, or create a new one. You should not define any "Content-XXX" headers by calling this method, because these headers use their own specific methods. Example of use:

```
$r->header_out("WWW-Authenticate" => "Basic");

$val = $r->header_out($key);
```

$r->err_headers_out
: The `$r->`err_headers_out method will return a `%hash` of server response headers. This can be used to initialize a perl hash, or one could use the `$r->`*err_header_out()* method (described below) to retrieve or set a specific header value directly.

The difference between headers_out and err_headers_out is that the latter are printed even on error, and persist across internal redirects (so the headers printed for ErrorDocument handlers will have them).

$r->err_header_out($header, [$value])
: Change the value of an error response header, or create a new one. These headers are used if the status indicates an error.

```
$r->err_header_out("Warning" => "Bad luck");

$val = $r->err_header_out($key);
```

$r->no_cache($boolean)
: This is a flag that indicates that the data being returned is volatile and the client should be told not to cache it.

$r->write_client(@list_of_data)
: Send data to the client. Unless you know what you are doing, you should only call this method after you have called `$r->`send_http_header.

$r->print()
: This method sends data to the client with `$r-`write_client>, but first sets a timeout before sending with `$r-`hard_timeout>.

$r->send_fd($filehandle)
: Send the contents of a file to the client. Can for instance be used like this:

```
open(FILE, $r->filename) || return 404;
$r->send_fd(FILE);
close(FILE);
```

$r->internal_redirect_handler($newplace)
: Redirect to a location in the server's namespace without telling the client. For instance:

```
$r->internal_redirect_handler("/home/sweet/home.html");
```

SERVER CORE FUNCTIONS

$r->soft_timeout($message)

$r->hard_timeout($message)

$r->kill_timeout

$r->reset_timeout

(Documentation borrowed from http_main.h)

There are two functions which modules can call to trigger a timeout (with the per-virtual-server timeout duration); these are hard_timeout and soft_timeout.

The difference between the two is what happens when the timeout expires (or earlier than that, if the client connection aborts) --- a soft_timeout just puts the connection to the client in an ''aborted'' state, which will cause http_protocol.c to stop trying to talk to the client, but otherwise allows the code to continue normally. *hard_timeout()*, by contrast, logs the request, and then aborts it completely --- *longjmp()*ing out to the *accept()* loop in http_main. Any resources tied into the request's resource pool will be cleaned up; everything that isn't will leak.

soft_timeout() is recommended as a general rule, because it gives your code a chance to clean up. However, *hard_timeout()* may be the most convenient way of dealing with timeouts waiting for some external resource other than the client, if you can live with the restrictions.

When a hard timeout is in scope, critical sections can be guarded with *block_alarms()* and *unblock_alarms()* --- these are declared in alloc.c because they are most often used in conjunction with routines to allocate something or other, to make sure that the cleanup does get registered before any alarm is allowed to happen which might require it to be cleaned up; they * are, however, implemented in http_main.c.

kill_timeout() will disarm either variety of timeout.

reset_timeout() resets the timeout in progress.

CGI SUPPORT

We also provide some methods that make it easier to support the CGI type of interface.

$r->cgi_env

Return a `%hash` that can be used to set up a standard CGI environment. Typical usage would be:

```
%ENV = $r->cgi_env
```

NOTE: The `$ENV{GATEWAY_INTERFACE}` is set to `'CGI-Perl/1.1'` so you can say:

```
if($ENV{GATEWAY_INTERFACE} =~ /^CGI-Perl/) {
    #do mod_perl stuff
}
else {
   #do normal CGI stuff
}
```

When given a key => value pair, this will set an environment variable.

```
$r->cgi_env(REMOTE_GROUP => "camels");
```

$r->cgi_var($key);

Calls `$r->`*cgi_env* ($key) in a scalar context to prevent the mistake of calling in a list context.

```
my $doc_root = $r->cgi_env('DOCUMENT_ROOT');
```

$r->send_cgi_header()

Take action on certain headers including *Status:*, *Location:* and *Content-type:* just as mod_cgi does, then calls `$r->`*send_http_header()*. Example of use:

```
$r->send_cgi_header("
Location: /foo/bar
Content-type: text/html

");
```

ERROR LOGGING

The following methods can be used to log errors.

$r->log_reason($message, $file)

The request failed, why?? Write a message to the server's errorlog.

```
$r->log_reason("Because I felt like it", $r->filename);
```

$r->log_error($message)

Uh, oh. Write a message to the server's errorlog.

```
$r->log_error("Some text that goes in the error_log");
```

UTILITY FUNCTIONS

Apache::unescape_url($string)

Handy function for unescapes.

Apache::perl_hook($hook)

Test to see if a callback hook is enabled

```
for (qw(Access Authen Authz Fixup HeaderParser Log Trans Type)) {
   print "$_ hook enabled\n" if Apache::perl_hook($_);
}
```

SEE ALSO

perl (1), *Apache::Constants* (3), *Apache::Registry* (3), *Apache::CGI* (3), *Apache::Debug* (3), *Apache::Options* (3)

AUTHORS

Gisle Aas <aas@sn.no> and Doug MacEachern <dougm@osf.org>

NAME

Apache::AccessLimitNum – Limit user access by number of requests

SYNOPSIS

```
#server config file

PerlModule Apache::AccessLimitNum

#access control directives

#use any authentication module
AuthName SomeRealm
Auth[DBM]UserFile /path/to/password/file

PerlAccessHandler Apache::AccessLimitNum::handler
PerlSetVar        AccessLimitNum  100
PerlSetVar        AccessLimitFile /path/to/limit/file

<Limit GET>
require valid-user #or some such
</Limit>
```

DESCRIPTION

Decide if an authenticated user has exceeded an access limit for the requested realm, if so, forbid access.

AccessLimitFile is a dbm file consisting of *username* = *number* value pairs. This file must be writeable by httpd server process.

SEE ALSO

Apache (3)

AUTHOR

Doug MacEachern <dougm@osf.org>

NAME

Apache::Authen − Building blocks for mod_perl PerlAuthenHandler's

SYNOPSIS

use Apache::Authen ()

DESCRIPTION

Building blocks for mod_perl PerlAuthenHandler's

METHODS

Apache::Authen->check($r, \%attr)

This method looks up the username and password via HTTPD::UserAdmin testing for a valid password if user is found. Returns `OK` upon success, otherwise returns `AUTH_REQUIRED`. `$r` is a request_rec object blessed into the the *Apache* manpage class. `\%attr` is a hash reference passed to HTTPD::UserAdmin->new.

SEE ALSO

Apache (3), *HTTPD::UserAdmin* (3), *Apache::AuthenDBI* (3)

AUTHOR

Doug MacEachern <dougm@osf.org>

NAME

Apache::AuthenDBI − Authenticate via Perl DBI

SYNOPSIS

```
#httpd.conf or srm.conf

PerlModule Apache::AuthenDBI

#.htaccess
AuthName DBI
AuthType Basic

#authenticate via DBI
PerlAuthenHandler Apache::AuthenDBI

PerlSetVar AuthDBIDB     dbname
PerlSetVar AuthDBIUser   username
PerlSetVar AuthDBIAuth   password
PerlSetVar AuthDBIDriver Oracle
#DBI->connect(qw(AuthDBIDB AuthDBIUser AuthDBIAuth AuthDBIDriver))

PerlSetVar AuthDBIUserTable www_users
PerlSetVar AuthDBINameField user
PerlSetVar AuthDBIPasswordField password
```

<Limit GET POST> require valid-user </Limit>

DESCRIPTION

Apache::AuthenDBI is a replacement for mod_auth_dbi, allowing the apache server to authenticate via Perl's DBI.

HANDLERS

Apache::AuthenDBI::handler

This handler authenticates against a database such as Oracle, DB2, Sybase, and others supported by the DBI module. For supported drivers see: http://www.hermetica.com/technologia/DBI

This handler users the *HTTPD::UserAdmin* manpage to lookup the username and password. This may change.

SEE ALSO

Apache (3), *HTTPD::UserAdmin* (3), *DBI* (3)

AUTHOR

Doug MacEachern <dougm@osf.org>

NAME

Apache::AuthzAge − Authorize based on age

SYNOPSIS

```
#server config file

PerlModule Apache::AuthzAge

#access control directives

#use standard authentication modules
AuthName SomeRealm
Auth[DBM]UserFile /path/to/password/file

PerlAuthzHandler Apache::AuthzAge::handler
PerlSetVar       UserAgeFile  /path/to/dbm_file

#user must be at least 21
<Limit GET>
require age 21
</Limit>
```

DESCRIPTION

Decide if an authenticated user is authorized to complete a request based on age.

UserAgeFile is a dbm file consisting of *username* = *age* value pairs.

SEE ALSO

Apache (3)

AUTHOR

Doug MacEachern <dougm@osf.org>

NAME

Apache::CGI – Make things work with CGI.pm against Perl-Apache API

SYNOPSIS

```
require Apache::CGI;

my $q = new Apache::CGI;

$q->print($q->header);

#do things just like you do with CGI.pm
```

DESCRIPTION

When using the Perl-Apache API, your applications are faster, but the enviroment is different than CGI. This module attempts to set-up that environment as best it can.

SEE ALSO

perl (1), *Apache* (3), *CGI* (3)

AUTHOR

Doug MacEachern <dougm@osf.org>, hacked over by Andreas König <a.koenig@mind.de>

NAME

Apache::Constants – Constants defined in httpd.h

SYNOPSIS

```
use Apache::Constants;
use Apache::Constants ':common'; #OK,DECLINED,etc.
```

DESCRIPTION

Server constants used by apache modules are defined in **httpd.h**, this module gives Perl access to those constants.

AUTHORS

Gisle Aas <aas@sn.no>, Doug MacEachern <dougm@osf.org> and h2xs

NAME

Apache::Debug – Utilities for debugging embedded perl code

SYNOPSIS

```
use Apache::Debug ();

Apache::Debug::dump($r, SERVER_ERROR, "Uh Oh!");
```

DESCRIPTION

This module sends what may be helpful debugging info to the client rather that the error log.

NAME

Apache::MsqlProxy − Translate URI's into mSQL database queries

SYNOPSIS

```
 #httpd.conf or srm.conf
 PerlTransHandler Apache::MsqlProxy::translate
 PerlHandler      Apache::MsqlProxy::handler
 PerlModule       Apache::MsqlProxy
```

Configure your browser's HTTP proxy to point at the host running Apache configured with this module:

```
  http://hostname.domain/
```

When connecting to the server via normal HTTP (not proxy), URLs are not translated.

URL's are translated as follows:

```
 http://hostname/database_name/table_name
```

Connect to **hostname** via TCP, select database **database_name**, query table <table_name> with:

```
 SELECT * from table_name

 http://hostname/database_name/table_name?login_name=dougm
```

Same as above with query:

```
 SELECT login_name from table where login_name=dougm

 http://hostname/database_name/table_name/*?login_name=dougm
```

Same as above with query:

```
 SELECT * from table where login_name=dougm
```

Of course,

```
 http:///database_name/table_name
```

A null hostname connects via UNIX socket

```
 http://hostname:9876/database_name/table_name
```

Connect via TCP to hostname and port 9876

DESCRIPTION

This module is meant as an example to show how one can use Apache + mod_perl to handle HTTP proxy requests, or simply translating a URL.

It may not be very useful other than as an example, but feel free to change that.

SEE ALSO

Apache (3), *Msql* (3)

AUTHOR

Doug MacEachern <dougm@osf.org>

NAME

Apache::Options – OPT_* defines from httpd_core.h

DESCRIPTION

The **Apache::Options** module will export the following bitmask constants:

```
OPT_NONE
OPT_INDEXES
OPT_INCLUDES
OPT_SYMLINKS
OPT_EXECCGI
OPT_UNSET
OPT_INCNOEXEC
OPT_SYM_OWNER
OPT_MULTI
OPT_ALL
```

These constants can be used to check the return value from Apache->request->*allow_options()* method.

This module is simply a stub which imports from the *Apache::Constants* manpage, just as if you had said `use Apache::Constants ':options';`.

SEE ALSO

the *Apache* manpage, the *Apache::Constants* manpage

NAME

Apache::Registry – Run (mostly) unaltered CGI scripts through mod_perl

SYNOPSIS

```
 #in srm.conf

 PerlAlias /perl/ /perl/apache/scripts/ #optional
 PerlModule Apache::Registry

 <Location /perl>
 SetHandler perl-script
 PerlHandler Apache::Registry
 ...
 </Directory>
```

DESCRIPTION

URIs in the form of:
 http://www.host.com/perl/file.pl

Will be compiled as the body of a perl subroutine and executed. Each server process or 'child' will compile the subroutine once and store it in memory until the file is updated on the disk.

The file looks much like a ''normal'' script, but it is compiled or 'evaled' into a subroutine.

Here's an example:

```
 my $r = Apache->request;
 $r->content_type("text/html");
 $r->send_http_header;
 $r->print("Hi There!");
```

Apache::Registry::handler will preform the same checks as mod_cgi before running the script.

DEBUGGING

You may set the debug level with the `$Apache::Registry::Debug` bitmask

```
 1 => log recompile in errorlog
 2 => Apache::Debug::dump in case of $@
 4 => trace pedantically

=head1 SEE ALSO
```

perl (1), *Apache* (3), *Apache::Debug* (3)

AUTHORS

Andreas Koenig <andreas.koenig@franz.ww.tu-berlin.de> and Doug MacEachern <dougm@osf.org>

NAME

Apache::SSI – Implement Server Side Includes in Perl

SYNOPSIS

wherever you choose:

<Files *.phtml> SetHandler perl-script PerlHandler Apache::SSI </Files>

You may wish to subclass Apache::SSI for your own extentions

```
package MySSI;
use Apache::SSI ();
@ISA = qw(Apache::SSI);

#embedded syntax:
#<!--#something cmd=doit -->
sub something {
   my($self, $attr) = @_;
   my $cmd = $attr->{cmd};
   ...
   return $a_string;
}
```

DESCRIPTION

Apache::SSI implements the functionality of mod_include for handling server-parsed html documents. Each ‘‘command’’ or element is handled by an Apache::SSI method of the same name. attribute=value pairs are parsed and passed to the method in an anonymous hash.

This module supports the same directives as mod_include, see it's documentation for commands and syntax.

In addition, Apache::SSI supports the following directives:

perlsub
This directive calls a perl subroutine:

```
Hello user from <!--#perlsub sub=remote_host -->
```

CAVEATS

This module is not complete, it does not provide the full functionality of mod_include.

There is no support for xssi directives.

SEE ALSO

mod_include, *mod_perl* (3), *HTML::TreeBuilder* (3), *perl* (1), *Apache* (3)

AUTHOR

Doug MacEachern <dougm@osf.org>

NAME

Apache::Status – Embedded interpreter status information

DESCRIPTION

The **Apache::Status** module provides some information about the status of the Perl interpreter embedded in the server.

Configure like so:

```
<Location /perl-status>
SetHandler  perl-script
PerlHandler Apache::Status
</Location>
```

PREREQUISITES

The *Devel::Symdump* module must be installed:

```
perl -MCPAN -e 'install "Devel::Symdump"'
```

Or fetch from: http://www.perl.com/cgi-bin/cpan_mod?module=Devel::Symdump

SEE ALSO

perl (1), *Apache* (3), *Devel::Symdump* (3)

AUTHOR

Doug MacEachern <dougm@osf.org>

NAME

Apache::TieHandle − tie stdio to Apache's i/o methods

SYNOPSIS

```
 use Apache::TieHandle ();

 tie *STDOUT => 'Apache::TieHash';

 print "Content-type: text/html\n\n" .
      "Ah, just like CGI\n";
```

DESCRIPTION

This module tie's stdio filehandles to Apache's i/o methods.

*** NOTE *** There is no need to use this module unless your version of Perl is <= 5.003 You must apply Sven Verdoolaege's patch 'pp_hot.patch' for it to work

AUTHOR

Well, Doug MacEachern wrote it, but it wouldn't be possible without the efforts of Larry Wall and the fearless perl5-porters who implemented tie'd filehandles...

NAME

Bundle::Apache − Install Apache mod_perl and related modules

SYNOPSIS

```
perl -MCPAN -e 'install Bundle::Apache'
```

CONTENTS

Apache − Perl interface to Apache server API

ExtUtils::Embed − Needed to build httpd

LWP::UserAgent − Web client to run mod_perl tests

HTML::TreeBuilder − Used for Apache::SSI

Devel::Symdump − Symbol table browsing with Apache::Status

HTTPD::UserAdmin − Apache::Authen stuff

CGI − CGI.pm

Apache::ePerl − Mark Imbriaco's mod_perl adaptation of 'ePerl'

DESCRIPTION

This bundle contains modules used by Apache mod_perl.

Asking CPAN.pm to install a bundle means to install the bundle itself along with all the modules contained in the CONTENTS section above. Modules that are up to date are not installed, of course.

AUTHOR

Doug MacEachern

NAME

CGI::Switch – Try more than one constructors and return the first object available

SYNOPSIS

```
use CGISwitch;

 -or-

use CGI::Switch This, That, CGI::XA, Foo, Bar, CGI;

my $q = new CGI::Switch;
```

DESCRIPTION

Per default the *new()* method tries to call *new()* in the three packages Apache::CGI, CGI::XA, and CGI. It returns the first CGI object it succeeds with.

The import method allows you to set up the default order of the modules to be tested.

SEE ALSO

perl (1), *Apache* (3), *CGI* (3), *CGI::XA* (3)

AUTHOR

Andreas König <a.koenig@mind.de>

NAME

CGI::XA − Clone of CGI.pm with less backwards compatibility and less namespace pollution

SYNOPSIS

```
  use CGI::XA;
  # then follow the CGI.pm docs and see if it does the same.
```

BE AWARE, THIS IS ALPHA SOFTWARE

It's main purpose is to start a discussion about CGI.pm. Maybe parts of this will be folded back to CGI.pm, and then probably this module won't be developed any further.

DESCRIPTION

I have started with major hacks on top of Lincoln's version 2.23 of CGI.pm in order to get rid of both AUTOLOADING and uncontrolled global variables.

I release this package as CGI::XA (which stands for ‘‘for Apache") for a limited audience as a test case.

This software is alpha software and it is not clear if it will be supported for a longer time. My preferred solution would be, Lincoln accepts (most of) the changes and continues with his excellent work.

where are the main differences?

```
    No AUTOLOAD
    use strict clean
    no exports
    no cgi−lib.pl compatibility
    no "Q" namespace
    abandoned the rearrange method in several places
```

DOCUMENTATION

is in CGI.pm with which we are quite compatible with. See comments in the code. If you could supply test cases for methods that are not depending on a browser, we'd be glad to hear from you.

AUTHOR

Lincoln D. Stein is th author of the original code which was the basis for this development. Responsibility is with Andreas König <andreas.koenig@mind.de>

SEE ALSO

perl (1), *Apache* (3), *Apache::Switch* (3).

NAME

LWP − Library for WWW access in Perl

DESCRIPTION

Libwww-perl is a collection of Perl modules which provides a simple and consistent programming interface (API) to the World-Wide Web. The main focus of the library is to provide classes and functions that allow you to write WWW clients, thus libwww-perl said to be a WWW client library. The library also contain modules that are of more general use.

The main architecture of the library is object oriented. The user agent, requests sent and responses received from the WWW server are all represented by objects. This makes a simple and powerful interface to these services. The interface should be easy to extend and customize for your needs.

The main features of the library are:

- Contains various reusable components (modules) that can be used separately or together.
- Provides an object oriented model of HTTP−style communication. Within this framework we currently support access to http, gopher, ftp, news, file, and mailto resources.
- The library be used through the full object oriented interface or through a very simple procedural interface.
- Support the basic and digest authorization schemes.
- Transparent redirect handling.
- Supports access through proxy servers.
- URL handling (both absolute and relative URLs are supported).
- A parser for *robots.txt* files and a framework for constructing robots.
- An experimental HTML parser and formatters (for PostScript and plain text).
- The library can cooperate with Tk. A simple Tk-based GUI browser called 'tkweb' is distributed with the Tk extension for perl.
- An implementation of the HTTP content negotiation algorithm that can be used both in protocol modules and in server scripts (like CGI scripts).
- A simple command line client application called `lwp−request`.

HTTP STYLE COMMUNICATION

The libwww-perl library is based on HTTP style communication. This section try to describe what that means.

Let us start with this quote from the HTTP specification document <URL:http://www.w3.org/pub/WWW/Protocols/>:

> The HTTP protocol is based on a request/response paradigm. A client establishes a connection with a server and sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME−like message containing request modifiers, client information, and possible body content. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME−like message containing server information, entity meta-information, and possible body content.

What this means to libwww-perl is that communication always take place through these steps: First a *request* object is created and configured. This object is then passed to a server and we get a *response* object in return that we can examine. A request is always independent of any previous requests, i.e. the service is stateless. The same simple model is used for any kind of service we want to access.

For example, if we want to fetch a document from a remote file server, then we send it a request that contains a name for that document and the response will contain the document itself. If we access a search engine, then the content of the request will contain the query parameters and the response will contain the query result. If we want to send a mail message to somebody then we send a request object which contains our message to the mail server and the response object will contain an acknowledgment that tells us that the message has been accepted and will be forwarded to the *recipient* (s).

It is as simple as that!

The Request Object

The request object has the class name `HTTP::Request` in libwww-perl. The fact that the class name use `HTTP::` as a name prefix only implies that we use the HTTP model of communication. It does not limit the kind of services we can try to pass this *request* to. For instance, we will send `HTTP::Requests` both to ftp and gopher servers, as well as to the local file system.

The main attributes of the request objects are:

- The **method** is a short string that tells what kind of request this is. The most used methods are **GET**, **PUT**, **POST** and **HEAD**.
- The **url** is a string denoting the protocol, server and the name of the "document" we want to access. The **url** might also encode various other parameters.
- The **headers** contain additional information about the request and can also used to describe the content. The headers is a set of keyword/value pairs.
- The **content** is an arbitrary amount of data.

The Response Object

The request object has the class name `HTTP::Response` in libwww-perl. The main attributes of objects of this class are:

- The **code** is a numerical value that encode the overall outcome of the request.
- The **message** is a short (human readable) string that corresponds to the *code*.
- The **headers** contain additional information about the response and they also describe the content.
- The **content** is an arbitrary amount of data.

Since we don't want to handle all possible *code* values directly in our programs, the libwww-perl response object have methods that can be used to query what kind of response this is. The most commonly used response classification methods are:

is_success()
: The request was was successfully received, understood or accepted.

is_error()
: The request failed. The server or the resource might not be available, access to the resource might be denied or other things might have failed for some reason.

The User Agent

Let us assume that we have created a *request* object. What do we actually do with it in order to receive a *response*?

The answer is that you pass it on to a *user agent* object and this object will take care of all the things that need to be done (low-level communication and error handling). The user agent will give you back a *response* object. The user agent represents your application on the network and it provides you with an interface that can accept *requests* and will return *responses*.

You should think about the user agent as an interface layer between your application code and the network. Through this interface you are able to access the various servers on the network.

The libwww-perl class name for the user agent is `LWP::UserAgent`. Every libwww-perl application that wants to communicate should create at least one object of this kind. The main method provided by this object is *request()*. This method takes an `HTTP::Request` object as argument and will (eventually) return a `HTTP::Response` object.

The user agent has many other attributes that lets you configure how it will interact with the network and with your application code.

- The **timeout** specify how much time we give remote servers in creating responses before the library disconnect and creates an internal *timeout* response.
- The **agent** specify the name that your application should use when it presents itself on the network.
- The **from** attribute can be set to the e-mail address of the person responsible for running the application. If this is set, then the address will be sent to the servers with every request.
- The **use_alarm** specify if it is OK for the user agent to use the *alarm*(2) system to implement timeouts.
- The **use_eval** specify if the agent should raise an exception (`die` in perl) if an error condition occur.
- The **parse_head** specify whether we should initialize response headers from the <head> section of HTML documents.
- The **proxy** and **no_proxy** specify if and when communication should go through a proxy server. <URL:http://www.w3.org/pub/WWW/Proxies/>
- The **credentials** provide a way to set up user names and passwords that is needed to access certain services.

Many applications would want even more control over how they interact with the network and they get this by specializing the `LWP::UserAgent` by sub-classing. The library provide a specialization called `LWP::RobotUA` that is used by robot applications.

An Example

This example shows how the user agent, a request and a response are represented in actual perl code:

```
# Create a user agent object
use LWP::UserAgent;
$ua = new LWP::UserAgent;
$ua->agent("AgentName/0.1 " . $ua->agent);

# Create a request
my $req = new HTTP::Request POST => 'http://www.perl.com/cgi-bin/BugGlimpse';
$req->content_type('application/x-www-form-urlencoded');
$req->content('match=www&errors=0');

# Pass request to the user agent and get a response back
my $res = $ua->request($req);

# Check the outcome of the response
if ($res->is_success) {
    print $res->content;
} else {
    print "Bad luck this time\n";
}
```

The `$ua` is created once when the application starts up. New request objects are normally created for each request sent.

NETWORK SUPPORT

This section goes through the various protocol schemes and describe the HTTP style methods that are supported and the headers that might have any effect.

For all requests, a "User-Agent" header is added and initialized from the `$ua`->agent value before the request is handed to the network layer. In the same way, a "From" header is initialized from the `$ua`->from value.

For all responses, the library will add a header called "Client-Date". This header will encode the time when the response was received by your application. This format and semantics of the header is just like the server created "Date" header.

HTTP Requests

HTTP request are really just handed off to an HTTP server and it will decide what happens. Few servers implement methods beside the usual "GET", "HEAD", "POST" and "PUT" but CGI–scripts can really implement any method they like.

If the server is not available then the library will generate an internal error response.

The library automatically adds a "Host" and a "Content-Length" header to the HTTP request before it is sent over the network.

For GET request you might want to add the "If-Modified-Since" header to make the request conditional.

For POST request you should add the "Content-Type" header. When you try to emulate HTML <FORM> handling you should usually let the value of the "Content-Type" header be "application/x-www-form-urlencoded". See the *lwpcook* manpage for examples of this.

The libwww-perl HTTP implementation currently support the HTTP/1.0 protocol. HTTP/0.9 servers are also handled correctly.

The library allows you to access proxy server through HTTP. This means that you can set up the library to forward all types of request through the HTTP protocol module. See the *LWP::UserAgent* manpage for documentation of this.

FTP Requests

The library currently support GET, HEAD and PUT requests. GET will retrieve a file or a directory listing from an FTP server. PUT will store a file on a ftp server.

You can specify a ftp account for servers that want this in addition user name and password. This is specified by passing an "Account" header in the request.

User name/password can be specified using basic authorization or be encoded in the URL. Bad logins return an UNAUTHORIZED response with "WWW–Authenticate: Basic" and can be treated as basic authorization for HTTP.

The library support ftp ASCII transfer mode by specifying the "type=a" parameter in the URL.

Directory listings are by default returned unprocessed (as returned from the ftp server) with the content media type reported to be "text/ftp-dir-listing". The `File::Listing` module provide functionality for parsing of these directory listing.

The ftp module is also able to convert directory listings to HTML and this can be requested via the standard HTTP content negotiation mechanisms (add an "Accept: text/html" header in the request if you want this).

The normal file retrievals, the "Content-Type" is guessed based on the file name suffix. See the *LWP::MediaTypes* manpage.

The "If-Modified-Since" header is not honored yet.

Example:

```
$req = HTTP::Request->new(GET => 'ftp://me:passwd@ftp.some.where.com/');
$req->header(Accept => "text/html, */*;q=0.1");
```

News Requests

Access to the USENET News system is implemented through the NNTP protocol. The name of the news server is obtained from the NNTP_SERVER environment variable and defaults to ‘‘news’’. It is not possible to specify the hostname of the NNTP server in the news:–URLs.

The library support GET and HEAD to retrieve news articles through the NNTP protocol. You can also post articles to newsgroups by using (surprise!) the POST method.

GET on newsgroups is not implemented yet.

Examples:

```
$req = HTTP::Request->new(GET => 'news:abc1234@a.sn.no');

$req = HTTP::Request->new(POST => 'news:comp.lang.perl.test');
$req->header(Subject => 'This is a test',
             From    => 'me@some.where.org');
$req->content(<<EOT);
This is the content of the message that we are sending to
the world.
EOT
```

Gopher Request

The library supports the GET and HEAD method for gopher request. All request header values are ignored. HEAD cheats and will return a response without even talking to server.

Gopher menus are always converted to HTML.

The response ‘‘Content-Type’’ is generated from the document type encoded (as the first letter) in the request URL path itself.

Example:

```
$req = HTTP::Request->new('GET', 'gopher://gopher.sn.no/');
```

File Request

The library supports GET and HEAD methods for file requests. The ‘‘If-Modified-Since’’ header is supported. All other headers are ignored. The *host* component of the file URL must be empty or set to ‘‘localhost’’. Any other *host* value will be treated as an error.

Directories are always converted to an HTML document. For normal files, the ‘‘Content-Type’’ and ‘‘Content-Encoding’’ in the response are guessed based on the file suffix.

Example:

```
$req = HTTP::Request->new(GET => 'file:/etc/passwd');
```

Mailto Request

You can send (aka ‘‘POST") mail messages using the library. All headers specified for the request are passed on to the mail system. The ‘‘To’’ header is initialized from the mail address in the URL.

Example:

```
$req = HTTP::Request->new(POST => 'mailto:libwww-perl-request@ics.uci.edu');
$req->header("Subject", "subscribe");
$req->content("Please subscribe me to the libwww-perl mailing list!\n");
```

OVERVIEW OF CLASSES AND PACKAGES

This table should give you a quick overview of the classes provided by the library. Indentation shows class inheritance.

```
LWP::MemberMixin   -- Access to member variables of Perl5 classes
  LWP::UserAgent   -- WWW user agent class
    LWP::RobotUA   -- When developing a robot applications
  LWP::Protocol          -- Interface to various protocol schemes
    LWP::Protocol::http  -- http:// access
    LWP::Protocol::file  -- file:// access
    LWP::Protocol::ftp   -- ftp:// access
    ...

LWP::Socket        -- Socket creation and IO

HTTP::Headers      -- MIME/RFC822 style header (used by HTTP::Message)
HTTP::Message      -- HTTP style message
  HTTP::Request    -- HTTP request
  HTTP::Response   -- HTTP response
HTTP::Daemon       -- A HTTP server class

URI::URL           -- Uniform Resource Locators

WWW::RobotRules    -- Parse robots.txt files
  WWW::RobotRules::AnyDBM_File -- Persistent RobotRules

HTML::Parser       -- Parse HTML documents
  HTML::TreeBuilder-- Build a HTML syntax tree
  HTML::HeadParser -- Parse the <HEAD> section of a HTML document
  HTML::LinkExtor  -- Extract links from a HTML document
HTML::Element      -- Building block for the HTML::TreeBuilder
HTML::Formatter    -- Convert HTML syntax trees to readable formats
  HTML::FormatText -- Output is plain text
  HTML::FormatPS   -- Output is PostScript
```

The following modules provide various functions and definitions.

```
LWP                -- This file.  Library version number and documentation.
LWP::MediaTypes    -- MIME types configuration (text/html etc.)
LWP::Debug         -- Debug logging module
LWP::Simple        -- Simplified procedural interface for common functions
HTTP::Status       -- HTTP status code (200 OK etc)
HTTP::Date         -- Date parsing module for HTTP date formats
HTTP::Negotiate    -- HTTP content negotiation calculation
HTML::Entities     -- Expand or unexpand entities in HTML text
File::Listing      -- Parse directory listings
```

HTTP use the Base64 encoding at some places. The QuotedPrint module is just included to make the MIME:: collection more complete.

```
MIME::Base64       -- Base64 encoding/decoding routines
MIME::QuotedPrint  -- Quoted Printable encoding/decoding routines
```

The following modules does not have much to do with the World Wide Web, but are included just because I am lazy and did not bother to make separate distributions for them. Regard them as bonus, provided free for your pleasure.

```
Font::AFM          -- Parse Adobe Font Metric files
File::CounterFile  -- Persistent counter class
```

MORE DOCUMENTATION

All modules contain detailed information on the interfaces they provide. The the *lwpcook* manpage is the libwww-perl cookbook that contain examples of typical usage of the library. You might want to take a look at how the scripts `lwp-request`, `lwp-rget` and `lwp-mirror` are implemented.

BUGS

The library can not handle multiple simultaneous requests yet. The HTML:: modules are still experimental. Also, check out what's left in the TODO file.

ACKNOWLEDGEMENTS

This package owes a lot in motivation, design, and code, to the libwww-perl library for Perl 4, maintained by Roy Fielding <fielding@ics.uci.edu>.

That package used work from Alberto Accomazzi, James Casey, Brooks Cutter, Martijn Koster, Oscar Nierstrasz, Mel Melchner, Gertjan van Oosten, Jared Rhine, Jack Shirazi, Gene Spafford, Marc VanHeyningen, Steven E. Brenner, Marion Hakanson, Waldemar Kebsch, Tony Sanders, and Larry Wall; see the libwww-perl-0.40 library for details.

The primary architect for this Perl 5 library is Martijn Koster and Gisle Aas, with lots of help from Graham Barr, Tim Bunce, Andreas Koenig, Jared Rhine, and Jack Shirazi.

COPYRIGHT

AVAILABILITY

The latest version of this library is likely to be available from:

```
http://www.sn.no/libwww-perl/
```

The best place to discuss this code is on the <libwww-perl@ics.uci.edu> mailing list.

NAME

lwpcook – libwww-perl cookbook

DESCRIPTION

This document contain some examples that show typical usage of the libwww-perl library. You should consult the documentation for the individual modules for more detail.

All examples should be runnable programs. You can, in most cases, test the code sections by piping the program text directly to perl.

GET

It is very easy to use this library to just fetch documents from the net. The LWP::Simple module provides the *get()* function that return the document specified by its URL argument:

```
use LWP::Simple;
$doc = get 'http://www.sn.no/libwww-perl/';
```

or, as a perl one-liner using the *getprint()* function:

```
perl -MLWP::Simple -e 'getprint "http://www.sn.no/libwww-perl/"'
```

or, how about fetching the latest perl by running this:

```
perl -MLWP::Simple -e '
  getstore "ftp://ftp.sunet.se/pub/lang/perl/CPAN/src/latest.tar.gz",
           "perl.tar.gz"'
```

You will probably first want to find a CPAN site closer to you by running something like the following command:

```
perl -MLWP::Simple -e 'getprint "http://www.perl.com/perl/CPAN/CPAN.html"'
```

Enough of this simple stuff! The LWP object oriented interface gives you more control over the request sent to the server. Using this interface you have full control over headers sent and how you want to handle the response returned.

```
use LWP::UserAgent;
$ua = new LWP::UserAgent;
$ua->agent("$0/0.1 " . $ua->agent);
# $ua->agent("Mozilla/5.0") # pretend you are some very new Netscape browser

$req = new HTTP::Request 'GET' => 'http://www.sn.no/libwww-perl';
$req->header('Accept' => 'text/html');

# send request
$res = $ua->request($req);

# check the outcome
if ($res->is_success) {
   print $res->content;
} else {
   print "Error: " . $res->code . " " . $res->message;
}
```

The lwp-request program (alias GET) that is distributed with the library can also be used to fetch documents from WWW servers.

HEAD

If you just want to check if a document is present (i.e. the URL is valid) try to run code that looks like this:

```
use LWP::Simple;

if (head($url)) {
   # ok document exists
}
```

The *head()* function really returns a list of meta-information about the document. The first three values of the list returned are the document type, the size of the document, and the age of the document.

More control over the request or access to all header values returned require that you use the object oriented interface described for GET above. Just s/GET/HEAD/g.

POST

There is no simple interface for posting data to a WWW server. You must use the object oriented interface for this. The most common POST operation is to access a WWW form application:

```
use LWP::UserAgent;
$ua = new LWP::UserAgent;

my $req = new HTTP::Request 'POST','http://www.perl.com/cgi-bin/BugGlimpse';
$req->content_type('application/x-www-form-urlencoded');
$req->content('match=www&errors=0');

my $res = $ua->request($req);
print $res->as_string;
```

If your application has the key/value pairs to be posted in an associative array, then we can exploit the URI::URL module to create the content for the POST request message (it handles all the escaping issues):

```
%form = ( search => 'www', errors => 0 );

use URI::URL;
use LWP::UserAgent;
$ua = new LWP::UserAgent;

my $req = new HTTP::Request 'POST', 'http://www.perl.com/cgi-bin/BugGlimpse';
$req->content_type('application/x-www-form-urlencoded');

my $curl = url("http:");      # create an empty HTTP URL object
$curl->query_form(%form);
$req->content($curl->equery); # %form content as escaped query string

print $ua->request($req)->as_string;
```

The lwp-request program (alias POST) that is distributed with the library can also be used for posting data.

PROXIES

Some sites use proxies to go through fire wall machines, or just as cache in order to improve performance. Proxies can also be used for accessing resources through protocols not supported directly (or supported badly :-) by the libwww-perl library.

You should initialize your proxy setting before you start sending requests:

```
use LWP::UserAgent;
$ua = new LWP::UserAgent;
$ua->env_proxy; # initialize from environment variables
# or
$ua->proxy(ftp  => 'http://proxy.myorg.com');
$ua->proxy(wais => 'http://proxy.myorg.com');
$ua->no_proxy(qw(no se fi));

my $req = new HTTP::Request 'wais://xxx.com/';
print $ua->request($req)->as_string;
```

The LWP::Simple interface will call *env_proxy()* for you automatically. Applications that use the `$ua->`*env_proxy()* method will normally not use the `$ua->`*proxy()* and `$ua->`*no_proxy()* methods.

ACCESS TO PROTECTED DOCUMENTS

Documents protected by basic authorization can easily be accessed like this:

```
use LWP::UserAgent;
$ua = new LWP::UserAgent;
$req = new HTTP::Request GET => 'http://www.sn.no/secret/';
$req->authorization_basic('aas', 'mypassword');
print $ua->request($req)->as_string;
```

The other alternative is to provide a subclass of *LWP::UserAgent* that overrides the *get_basic_credentials()* method. Study the *lwp-request* program for an example of this.

MIRRORING

If you want to mirror documents from a WWW server, then try to run code similar to this at regular intervals:

```
use LWP::Simple;

%mirrors = (
   'http://www.sn.no/'              => 'sn.html',
   'http://www.perl.com/'           => 'perl.html',
   'http://www.sn.no/libwww-perl/' => 'lwp.html',
   'gopher://gopher.sn.no/'         => 'gopher.html',
);

while (($url, $localfile) = each(%mirrors)) {
   mirror($url, $localfile);
}
```

Or, as a perl one-liner:

```
perl -MLWP::Simple -e 'mirror("http://www.perl.com/", "perl.html")';
```

The document will not be transfered unless it has been updated.

LARGE DOCUMENTS

If the document you want to fetch is too large to be kept in memory, then you have two alternatives. You can instruct the library to write the document content to a file (second `$ua->`*request()* argument is a file name):

```
use LWP::UserAgent;
$ua = new LWP::UserAgent;
```

```
my $req = new HTTP::Request 'GET',
              'http://www.sn.no/~aas/perl/www/libwww-perl-5.00.tar.gz';
$res = $ua->request($req, "libwww-perl.tar.gz");
if ($res->is_success) {
   print "ok\n";
}
```

Or you can process the document as it arrives (second `$ua->`*request()* argument is a code reference):

```
use LWP::UserAgent;
$ua = new LWP::UserAgent;
$URL = 'ftp://ftp.unit.no/pub/rfc/rfc-index.txt';

my $expected_length;
my $bytes_received = 0;
$ua->request(HTTP::Request->new('GET', $URL),
             sub {
                 my($chunk, $res) = @_;
                 $bytes_received += length($chunk);
                 unless (defined $expected_length) {
                    $expected_length = $res->content_length || 0;
                 }
                 if ($expected_length) {
                      printf STDERR "%d%% - ",
                                100 * $bytes_received / $expected_length;
                 }
                 print STDERR "$bytes_received bytes received\n";

                 # XXX Should really do something with the chunk itself
                 # print $chunk;
             });
```

HTML FORMATTING

It is easy to convert HTML code to "readable" text.

```
use LWP::Simple;
use HTML::Parse;
print parse_html(get 'http://www.sn.no/libwww-perl/')->format;
```

PARSE URLS

To access individual elements of a URL, try this:

```
use URI::URL;
$host = url("http://www.sn.no/")->host;
```

or

```
use URI::URL;
$u = url("ftp://ftp.sn.no/test/aas;type=i");
print "Protocol scheme is ", $u->scheme, "\n";
print "Host is ", $u->host, " at port ", $u->port, "\n";
```

or even

```
use URI::URL;
my($host,$port) = (url("ftp://ftp.sn.no/test/aas;type=i")->crack)[3,4];
```

EXPAND RELATIVE URLS

This code reads URLs and print expanded version.

```
use URI::URL;
$BASE = "http://www.sn.no/some/place?query";
while (<>) {
   print url($_, $BASE)->abs->as_string, "\n";
}
```

We can expand URLs in an HTML document by using the parser to build a tree that we then traverse:

```
%link_elements =
(
 'a'    => 'href',
 'img'  => 'src',
 'form' => 'action',
 'link' => 'href',
);

use HTML::Parse;
use URI::URL;

$BASE = "http://somewhere/root/";
$h = parse_htmlfile("xxx.html");
$h->traverse(\&expand_urls, 1);

print $h->as_HTML;

sub expand_urls
{
   my($e, $start) = @_;
   return 1 unless $start;
   my $attr = $link_elements{$e->tag};
   return 1 unless defined $attr;
   my $url = $e->attr($attr);
   return 1 unless defined $url;
   $e->attr($attr, url($url, $BASE)->abs->as_string);
}
```

BASE URL

If you want to resolve relative links in a page you will have to determine which base URL to use. The HTTP::Response objects now has a *base()* method.

```
$BASE = $res->base;
```

NAME

lwp-download – fetch large files from the net

SYNOPSIS

```
lwp-download <url> [<local file>]
```

DESCRIPTION

The *lwp-download* program will down load the document specified by the URL given as the first command line argument to a local file. The local filename used to save the document is guessed from the URL unless specified as the second command line argument.

The *lwp-download* program is implemented using the *libwww-perl* library. It is better suited to down load big files than the *lwp-request* program because it does not store the file in memory. Another benefit is that it will keep you updated about it's progress and that you don't have any options to worry about.

EXAMPLE

Fetch the newest and greatest perl version:

```
$ lwp-download http://www.perl.com/CPAN/src/latest.tar.gz
Saving to 'latest.tar.gz'...
1.47 MB received in 22 seconds (68.7 KB/sec)
```

AUTHOR

Gisle Aas <gisle@aas.no>

NAME

lwp-mirror − Simple mirror utility for WWW

SYNOPSIS

```
lwp-mirror [-v] [-t timeout] <url> <local file>
```

DESCRIPTION

This program can be used to mirror a document from a WWW server. The document is only transfered if the remote copy is newer than the local copy. If the local copy is newer nothing happens.

Use the `−v` option to print the version number of this program.

The timeout value specified with the `−t` option. The timeout value is the time that the program will wait for response from the remote server before it fails. The default unit for the timeout value is seconds. You might append ‘‘m’’ or ‘‘h’’ to the timeout value to make it minutes or hours, repectively.

Because this program is implemented using the LWP library, it only supports the protocols that LWP supports.

SEE ALSO

the *lwp-request* manpage, the *LWP* manpage

AUTHOR

Gisle Aas <aas@a.sn.no>

NAME

lwp-request, GET, HEAD, POST − Simple WWW user agent

SYNOPSIS

```
 lwp-request [-eEdvhx] [-m method] [-b <base URL>] [-t <timeout>]
             [-i <if-modified-since>] [-c <content-type>] [-C <credentials>]
             [-p <proxy-url>] [-o <format>] <url>...
```

DESCRIPTION

This program can be used to send requests to WWW servers and your local file system. The request content for POST, PUT and CHECKIN methods is read from stdin. The content of the response is printed on stdout. Error messages are printed on stderr. The program returns a status value indicating the number of URLs that failed.

The options are:

-m <method>
: Set which method to use for the request. If this option is not used, then the method is derived from the name of the program.

-f
: Force request through, even if the program believes that the method is illegal. The server will probably reject the request.

-b <url>
: This URL will be used as the base URL for the URLs that the method is applied to. The base URL only takes effect for relative URLs. If you do not provide this option and the URLs are relative, then they will be treated as files in the local file system.

-t <timeout>
: Set the timeout value for the requests. The timeout is the amount of time that the program will wait for a response from the remote server before it fails. The default unit for the timeout value is seconds. You might append "m" or "h" to the timeout value to make it minutes or hours, respectively. The default timeout is '3m', i.e. 3 minutes.

-i <time>
: Set the If-Modified-Since header in the request. If *time* it the name of a file, use the modification timestamp for this file. If *time* is not a file, it is parsed as a literal date. Take a look at the *HTTP::Date* manpage for recogniced formats.

-c <content-type>
: Set the Content-Type for the request. This option is only allowed for requests that take a content, i.e. POST, PUT and CHECKIN. You can force methods to take content by using the `-f` option together with `-c`. The default Content-Type for POST is `application/x-www-form-urlencoded`. The default Content-type for the others is `text/plain`.

-p <proxy-url>
: Set the proxy to be used for the requests. The program also loads proxy settings from the environment. You can disable this with the `-P` option.

-C <username>:<password>
: Provide credentials for documents that are protected by Basic Authentication. If the document is protected and you did not specify the username and password with this option, then you will be prompted to provide these values.

The following options controls what is displayed by the program:

-u
: Print request method and absolute URL as requests are made.

-U
: Print requeset headers in addition to request method and absolute URL.

-s Print response status code. This option is always on for HEAD requests.

-S Print response status chain. This shows redirect and autorization requests that are handled by the library.

-e Print response headers. This option is always on for HEAD requests.

-d Do **not** print the content of the response.

-o <format>
Process HTML content in various ways before printing it. If the content type of the response is not HTML, then this option has no effect. The legal format values are; *text*, *ps*, *links*, *html* and *dump*.

If you specify the *text* format then the HTML will be formatted as plain latin1 text. If you specify the *ps* format then it will be formatted as Postscript.

The *links* format will output all links found in the HTML document. Relative links will be expanded to absolute ones.

The *html* format will reformat the HTML code and the *dump* format will just dump the HTML syntax tree.

-v Print the version number of the program and quit.

-h Print usage message and quit.

-x Extra debugging output.

Because this program is implemented using the LWP library, it will only support the protocols that LWP supports.

SEE ALSO

the *lwp-mirror* manpage, the *LWP* manpage

COPYRIGHT

AUTHOR

Gisle Aas <aas@a.sn.no>

NAME

lwp-rget − Retrieve WWW documents recursively

SYNOPSIS

```
lwp-rget [--verbose] [--depth=N] [--limit=N] [--prefix=URL] <URL>
lwp-rget --version
```

DESCRIPTION

This program will retrieve a document and store it in a local file. It will follow any links found in the document and store these documents as well, patching links so that they refer to these local copies. This process continues until there are no more unvisited links or the process is stopped by the one or more of the limits which can be controlled by the command line arguments.

This program is useful if you want to make a local copy of a collection of documents or want to do web reading off-line.

All documents are stored as plain files in the current directory. The file names chosen are derived from the last component of URL paths.

The options are:

--depth=*n*

Limit the recursive level. Embedded images are always loaded, even if they fall outside the *--depth*. This means that one can use *--depth=0* in order to fetch a single document together with all inline graphics.

The default depth is 5.

--limit=*n*

Limit the number of documents to get. The default limit is 50.

--prefix=*url_prefix*

Limit the links to follow. Only URLs that start the prefix string are followed.

The default prefix is set as the ''directory'' of the initial URL to follow. For instance if we start lwp-rget with the URL `http://www.sn.no/foo/bar.html`, then prefix will be set to `http://www.sn.no/foo/`.

Use `--prefix=''` if you don't want the fetching to be limited by any prefix.

--sleep=*n*

Sleep *n* seconds before retrieving each document. This options allows you to go slowly, not loading the server you visiting too much.

--verbose

Make more noise while running.

--quiet

Don't make any noise.

--version

Print program version number and quit.

--help

Print the usage message and quit.

Before the program exits the name of the file, where the initial URL is stored, is printed on stdout. All used filenames are also printed on stderr as they are loaded. This printing can be suppressed with the *--quiet* option.

SEE ALSO

the *lwp-request* manpage, the *LWP* manpage

AUTHOR

Gisle Aas <aas@sn.no>

NAME

File::CounterFile − Persistent counter class

SYNOPSIS

```
use File::CounterFile;
$c = new File::CounterFile "COUNTER", "aa00";

$id = $c->inc;
open(F, ">F$id");
```

DESCRIPTION

This module implements a persistent counter class. Each counter is represented by a separate file in the file system. File locking is applied, so multiple processes might try to access the same counters at the same time without risk of counter destruction.

You give the file name as the first parameter to the object constructor (`new`). The file is created if it does not exist.

If the file name does not start with "/" or ".", then it is interpreted as a file relative to `$File::CounterFile::DEFAULT_DIR`. The default value for this variable is initialized from the environment variable `TMPDIR`, or */usr/tmp* is no environment variable is defined. You may want to assign a different value to this variable before creating counters.

If you pass a second parameter to the constructor, that sets the initial value for a new counter. This parameter only takes effect when the file is created (i.e. it does not exist before the call).

When you call the `inc()` method, you increment the counter value by one. When you call `dec()` the counter value is decrementd. In both cases the new value is returned. The `dec()` method only works for numerical counters (digits only).

You can peek at the value of the counter (without incrementing it) by using the `value()` method.

The counter can be locked and unlocked with the `lock()` and `unlock()` methods. Incrementing and value retrieval is faster when the counter is locked, because we do not have to update the counter file all the time. You can query whether the counter is locked with the `locked()` method.

There is also an operator overloading interface to the File::CounterFile object. This means that you might use the `++` operator for incrementing the counter, `--` operator for decrementing and you can interpolate counters diretly into strings.

BUGS

It uses *flock*(2) to lock the counter file. This does not work on all systems. Perhaps we should use the File::Lock module?

INSTALLATION

Copy this file to the *File* subdirectory of your Perl 5 library directory (often */usr/local/lib/perl5*).

COPYRIGHT

AUTHOR

Gisle Aas <aas@sn.no>

NAME

parse_dir − parse directory listing

SYNOPSIS

```
use File::Listing;
for (parse_dir(`ls -l`)) {
    ($name, $type, $size, $mtime, $mode) = @$_;
    next if $type ne 'f'; # plain file
    #...
}

# directory listing can also be read from a file
open(LISTING, "zcat ls-lR.gz|");
$dir = parse_dir(\*LISTING, '+0000');
```

DESCRIPTION

The *parse_dir()* routine can be used to parse directory listings. Currently it only understand Unix `'ls -l'` and `'ls -lR'` format. It should eventually be able to most things you might get back from a ftp server file listing (LIST command), i.e. VMS listings, NT listings, DOS listings,...

The first parameter to *parse_dir()* is the directory listing to parse. It can be a scalar, a reference to an array of directory lines or a glob representing a filehandle to read the directory listing from.

The second parameter is the time zone to use when parsing time stamps in the listing. If this value is undefined, then the local time zone is assumed.

The third parameter is the type of listing to assume. The values will be strings like 'unix', 'vms', 'dos'. Currently only 'unix' is implemented and this is also the default value. Ideally, the listing type should be determined automatically.

The fourth parameter specify how unparseable lines should be treated. Values can be 'ignore', 'warn' or a code reference. Warn means that the perl *warn()* function will be called. If a code reference is passed, then this routine will be called and the return value from it will be incorporated in the listing. The default is 'ignore'.

Only the first parameter is mandatory. The *parse_dir()* prototype is ($;$$$).

The return value from *parse_dir()* is a list of directory entries. In scalar context the return value is a reference to the list. The directory entries are represented by an array consisting of [`$filename`, `$filetype`, `$filesize`, `$filetime`, `$filemode`]. The `$filetype` value is one of the letters 'f', 'd', 'l' or '?'. The `$filetime` value is converted to seconds since Jan 1, 1970. The `$filemode` is a bitmask like the mode returned by *stat()*.

CREDITS

Based on lsparse.pl (from Lee McLoughlin's ftp mirror package) and Net::FTP's parse_dir (Graham Barr).

NAME

Font::AFM − Interface to Adobe Font Metrics files

SYNOPSIS

```
use Font::AFM;
$h = new Font::AFM "Helvetica";
$copyright = $h->Notice;
$w = $h->Wx->{"aring"};
$w = $h->stringwidth("Gisle", 10);
$h->dump;  # for debugging
```

DESCRIPTION

This module implements the Font::AFM class. Objects of this class are initialised from an AFM−file and allows you to obtain information about the font and the metrics of the various glyphs in the font.

All measurements in AFM files are given in terms of units equal to 1/1000 of the scale factor of the font being used. To compute actual sizes in a document, these amounts should be multiplied by (scale factor of font)/1000.

The following methods are available:

new($fontname)
: Object constructor. Takes the name of the font as argument. It will croak if the font can not be found.

latin1_wx_table()
: Returns a 256 element array, where each element contains the width of the corresponding character in the iso-8859-1 character set.

stringwidth($string, [$fontsize])
: Returns the width of the string passed as argument. The string is assumed to be encoded in the iso-8859-1 character set. A second argument can be used to scale the width according to the font size.

FontName
: The name of the font as presented to the PostScript language `findfont` operator, for instance ‘‘Times-Roman’’.

FullName
: Unique, human-readable name for an individual font, for instance ‘‘Times Roman’’.

FamilyName
: Human-readable name for a group of fonts that are stylistic variants of a single design. All fonts that are member of such a group should have exactly the same `FamilyName`. Example of a family name is ‘‘Times’’.

Weight
: Human-readable name for the weight, or ‘‘boldness’’, attribute of a font. Exampes are `Roman`, `Bold`, `Light`.

ItalicAngle
: Angle in degrees counterclockwise from the vertical of the dominant vertical strokes of the font.

IsFixedPitch
: If the value is `true`, it indicated that the font is a fixed-pitch (monospaced) font.

FontBBox
: A string of four numbers giving the lower-left x, lower-left y, upper-right x, and upper-right y of the font bounding box. The font bounding box is the smallest rectangle enclosing the shape that would result if all the characters of the font were placed with their origins coincident, and then painted.

UnderlinePosition
: Recommended distance from the baseline for positioning underline stokes. This number is the y coordinate of the center of the stroke.

UnderlineThickness
Recommended stroke width for underlining.

Version
Version number of the font.

Notice
Trademark or copyright notice, if applicable.

Comment
Comments found in the AFM file.

EncodingScheme
The name of the standard encoding scheme for the font. Most Adobe fonts use the `AdobeStandard-Encoding`. Special fonts might state `FontSpecific`.

CapHeight
Usually the y-value of the top of the capital H.

XHeight
Typically the y-value of the top of the lowercase x.

Ascender
Typically the y-value of the top of the lowercase d.

Descender
Typically the y-value of the bottom of the lowercase p.

Wx
Returns a hash table that maps from glyph names to the width of that glyph.

BBox
Returns a hash table that maps from glyph names to bounding box information. The bounding box consist of 4 numbers: llx, lly, urx, ury.

dump
Dumps the content of the Font::AFM object to STDOUT. Might sometimes be useful for debugging.

The AFM specification can be found at:

```
ftp://ftp.adobe.com/pub/adobe/DeveloperSupport/TechNotes/PSfiles/5004.AFM_Spec.ps
```

ENVIRONMENT

METRICS Contains the path to seach for AFM−files. Format is as for the PATH environment variable. The default path built into this library is:

```
/usr/lib/afm:/usr/local/lib/afm:/usr/openwin/lib/fonts/afm/:.
```

BUGS

Kerning data and composite character data is not yet parsed. Ligature data is not parsed.

COPYRIGHT

NAME

HTML::AsSubs − functions that construct a HTML syntax tree

SYNOPSIS

```
use HTML::AsSubs;
$h = body(
          h1("This is the heading"),
          p("This is the first paragraph which contains a ",
            a({href=>'link.html'}, "link"),
            " and an ",
            img({src=>'img.gif', alt=>'image'}),
            "."
           ),
         );
print $h->as_HTML;
```

DESCRIPTION

This module exports functions that can be used to construct various HTML elements. The functions are named after the tags of the correponding HTML element and are all written in lower case. If the first argument is a *hash* then it will be used to initialize the attributes of this element. The remaining arguments are regarded as content.

ACKNOWLEDGEMENT

This module was inspired by the following message:

```
Date: Tue, 4 Oct 1994 16:11:30 +0100
Subject: Wow! I have a large lightbulb above my head!

Take a moment to consider these lines:

%OVERLOAD=( '""' => sub { join("", @{$_[0]}) } );

sub html { my($type)=shift; bless ["<$type>", @_, "</$type>"]; }

:-)  I *love* Perl 5!  Thankyou Larry and Ilya.

Regards,
Tim Bunce.

p.s. If you didn't get it, think about recursive data types: html(html())
p.p.s. I'll turn this into a much more practical example in a day or two.
p.p.p.s. It's a pity that overloads are not inherited. Is this a bug?
```

BUGS

The exported *link()* function overrides the builtin *link()* function. The exported *tr()* function must be called using &*tr* (...) syntax because it clashes with the builtin tr/../../ operator.

SEE ALSO

the *HTML::Element* manpage

NAME

HTML::Element – Class for objects that represent HTML elements

SYNOPSIS

```
require HTML::Element;
$a = new HTML::Element 'a', href => 'http://www.oslonett.no/';
$a->push_content("Oslonett AS");

$tag = $a->tag;
$tag = $a->starttag;
$tag = $a->endtag;
$ref = $a->attr('href');

$links = $a->extract_links();

print $a->as_HTML;
```

DESCRIPTION

Objects of the HTML::Element class can be used to represent elements of HTML. These objects have attributes and content. The content is an array of text segments and other HTML::Element objects. Thus a tree of HTML::Element objects as nodes can represent the syntax tree for a HTML document.

The following methods are available:

`$h` = HTML::Element->new('tag', 'attrname' => 'value',...)

The object constructor. Takes a tag name as argument. Optionally, allows you to specify initial attributes at object creation time.

`$h`–>*tag()*

Returns (optionally sets) the tag name for the element. The tag is always converted to lower case.

`$h`–>*starttag()*

Returns the complete start tag for the element. Including leading "<", trailing ">" and attributes.

`$h`–>*endtag()*

Returns the complete end tag. Includes leading "</" and the trailing ">".

`$h`–>*parent* ([$newparent])

Returns (optionally sets) the parent for this element.

`$h`–>*implicit* ([$bool])

Returns (optionally sets) the implicit attribute. This attribute is used to indicate that the element was not originally present in the source, but was inserted in order to conform to HTML strucure.

`$h`–>is_inside('tag',...)

Returns true if this tag is contained inside one of the specified tags.

`$h`–>*pos()*

Returns (and optionally sets) the current position. The position is a reference to a HTML::Element object that is part of the tree that has the current object as root. This restriction is not enforced when setting *pos()*, but unpredictable things will happen if this is not true.

$h->attr('attr', [$value])

Returns (and optionally sets) the value of some attribute.

$h->*content()*

Returns the content of this element. The content is represented as a reference to an array of text segments and references to other HTML::Element objects.

$h->*is_empty()*

Returns true if there is no content.

$h->insert_element($element, $implicit)

Inserts a new element at current position and updates *pos()* to point to the inserted element. Returns `$element`.

$h->push_content($element_or_text,...)

Adds to the content of the element. The content should be a text segment (scalar) or a reference to a HTML::Element object.

$h->*delete_content()*

Clears the content.

$h->*delete()*

Frees memory associated with the element and all children. This is needed because perl's reference counting does not work since we use circular references.

$h->traverse(\&callback, [$ignoretext])

Traverse the element and all of its children. For each node visited, the callback routine is called with the node, a startflag and the depth as arguments. If the `$ignoretext` parameter is true, then the callback will not be called for text content. The flag is 1 when we enter a node and 0 when we leave the node.

If the returned value from the callback is false then we will not traverse the children.

$h->*extract_links* ([@wantedTypes])

Returns links found by traversing the element and all of its children. The return value is a reference to an array. Each element of the array is an array with 2 values; the link value and a reference to the corresponding element.

You might specify that you just want to extract some types of links. For instance if you only want to extract <a href="..."> and <img src="..."> links you might code it like this:

```
for (@{ $e->extract_links(qw(a img)) }) {
    ($link, $linkelem) = @$_;
    ...
}
```

$h->*dump()*

Prints the element and all its children to STDOUT. Mainly useful for debugging. The structure of the document is shown by indentation (no end tags).

`$h->`*as_HTML()*

Returns a string (the HTML document) that represents the element and its children.

BUGS

If you want to free the memory assosiated with a tree built of HTML::Element nodes then you will have to delete it explicitly. The reason for this is that perl currently has no proper garbage collector, but depends on reference counts in the objects. This scheme fails because the parse tree contains circular references (parents have references to their children and children have a reference to their parent).

SEE ALSO

the *HTML::AsSubs* manpage

COPYRIGHT

AUTHOR

Gisle Aas <aas@sn.no>

NAME

decode – Expand HTML entities in a string

encode – Encode chars in a string using HTML entities

SYNOPSIS

```
use HTML::Entities;

$a = "V&aring;re norske tegn b&oslash;r &#230res";
decode_entities($a);
encode_entities($a, "\200-\377");
```

DESCRIPTION

The *decode_entities()* routine replaces valid HTML entities found in the string with the corresponding ISO–8859/1 character.

The *encode_entities()* routine replaces the characters specified by the second argument with their entity representation. The default set of characters to expand are control chars, high-bit chars and the ‘<’, ‘&’, ‘>’ and ‘"’ character.

Both routines modify the string passed in as the first argument and return it.

If you prefer not to import these routines into your namespace you can call them as:

```
use HTML::Entities ();
$encoded = HTML::Entities::encode($a);
$decoded = HTML::Entities::decode($a);
```

The module can also export the `%char2entity` and the `%entity2char` hashes which contains the mapping from all characters to the corresponding entities.

COPYRIGHT

AUTHOR

Gisle Aas <aas@sn.no>

NAME

HTML::FormatPS − Format HTML as postscript

SYNOPSIS

```
require HTML::FormatPS;
$html = parse_htmlfile("test.html");
$formatter = new HTML::FormatPS
                 FontFamily => 'Helvetica',
                 PaperSize  => 'Letter';
print $formatter->format($html);
```

DESCRIPTION

The HTML::FormatPS is a formatter that outputs PostScript code. Formatting of HTML tables and forms is not implemented.

You might specify the following parameters when constructing the formatter:

PaperSize
: What kind of paper should we format for. The value can be one of these: A3, A4, A5, B4, B5, Letter, Legal, Executive, Tabloid, Statement, Folio, 10x14, Quarto.

  The default is "A4".

PaperWidth
: The width of the paper in points. Setting PaperSize also defines this value.

PaperHeight
: The height of the paper in points. Setting PaperSize also defines this value.

LeftMargin
: The left margin in points.

RightMargin
: The right margin in points.

HorizontalMargin
: Both left and right margin at the same time. The default value is 4 cm.

TopMargin
: The top margin in points.

BottomMargin
: The bottom margin in points.

VerticalMargin
: Both top and bottom margin at the same time. The default value is 2 cm.

PageNo
: The parameter determines if we should put page numbers on the pages. The default is yes, so you have to set this value to 0 in order to suppress page numbers.

FontFamily
: The parameter specifies which family of fonts to use for the formatting. Legal values are "Courier", "Helvetica" and "Times". The default is "Times".

FontScale
: All fontsizes might be scaled by this factor.

Leading
: How much space between lines. This is a factor of the fontsize used for that line. Default is 0.1.

SEE ALSO

the *HTML::Formatter* manpage

COPYRIGHT

AUTHOR

Gisle Aas <aas@oslonett.no>

NAME

HTML::FormatText – Format HTML as text

SYNOPSIS

```
require HTML::FormatText;
$html = parse_htmlfile("test.html");
$formatter = new HTML::FormatText;
print $formatter->format($html);
```

DESCRIPTION

The HTML::FormatText is a formatter that outputs plain latin1 text. All character attributes (bold/italic/underline) are ignored. Formatting of HTML tables and forms is not implemented.

SEE ALSO

the *HTML::Formatter* manpage

COPYRIGHT

AUTHOR

Gisle Aas <aas@oslonett.no>

NAME

HTML::Formatter − Base class for HTML formatters

DESCRIPTION

HTML formatters are able to format a HTML syntax tree into various printable formats. Different formatters produce output for different output media. Common for all formatters are that they will return the formatted output when the *format()* method is called. *Format()* takes a HTML::Element as parameter.

SEE ALSO

the *HTML::FormatText* manpage, the *HTML::FormatPS* manpage, the *HTML::Element* manpage

COPYRIGHT

AUTHOR

Gisle Aas <aas@oslonett.no>

NAME

HTML::HeadParser – Parse <HEAD> section of a HTML document

SYNOPSIS

```
require HTML::HeadParser;
$p = HTML::HeadParser->new;
$p->parse($text) and  print "not finished";

$p->header('Title')          # to access <title>....</title>
$p->header('Content-Base')   # to access <base href="http://...">
$p->header('Foo')            # to access <meta http-equiv="Foo" content="...">
```

DESCRIPTION

The *HTML::HeadParser* is a specialized (and lightweight) *HTML::Parser* that will only parse the <HEAD>...</HEAD> section of a HTML document. The *parse()* and *parse_file()* methods will return a FALSE value as soon as a <BODY> element is found, and should not be called again after this.

The *HTML::HeadParser* constructor takes a *HTTP::Headers* object reference as argument. The parser will update this header object as the various head elements are recognized.

The following header fields are initialized from elements found in the <head> section of a HTML document:

Content-Base:
: The *Content-Base* header is initialized from the <base href=''..."> element.

Title:
: The *Title* header is initialized from the <title>...</title> element.

Isindex:
: The *Isindex* header will be added if there is a <isindex> element in the <head>. The header value is initialized from the *prompt* attribute if it is present.

X–Meta-Foo
: All <meta> elements will initialize headers with the prefix ''X–Meta-''. If the element contains a *http-equiv* attribute, then it will be honored as the header name.

EXAMPLES

```
$h = HTTP::Headers->new;
$p = HTML::HeadParser->new($h);
$p->parse(<<EOT);
<title>Stupid example</title>
<base href="http://www.sn.no/libwww-perl/">
Normal text starts here.
EOT
undef $p;
print $h->title;   # should print "Stupid example"
```

SEE ALSO

the *HTML::Parser* manpage, the *HTTP::Headers* manpage

COPYRIGHT

AUTHOR

Gisle Aas <aas@sn.no>

NAME

HTML::LinkExtor – Extract links from an HTML document

SYNPOSIS

```
require HTML::LinkExtor;
$p = HTML::LinkExtor->new(\&cb, "http://www.sn.no/");
sub cb {
    my($tag, %links);
    print "$tag @{[%links]}\n";
}
$p->parse_file("index.html");
```

DESCRIPTION

The *HTML::LinkExtor* (link extractor) is an HTML parser that takes a callback routine as parameter. This routine is then called as the various link attributes are recognized.

The *HTML::LinkExtor* is a subclass of *HTML::Parser*. This means that the document should be given to the parser by calling the `$p`–>*parse()* or `$p`–>*parse_file()* methods.

`$p` = HTML::LinkExtor->new([$callback[, `$base`]])

The constructor takes two optional argument. The first is a reference to a callback routine. It will be called as links are found. If a callback is not provided, then links are just accumulated internally and can be retrieved by calling the `$p`–>*links()* method. The `$base` is an optional base URL used to absolutize all URLs found.

The callback is called with the lowercase tag name as first argument, and then all link attributes as separate key/value pairs. All non-link attributes are removed.

`@links` = `$p`–>links

Return links found in the document as an array. Each array element contains an anonymous array with the follwing values:

```
[$tag, $attr => $url1, $attr2 => $url2,...]
```

Note that `$p`–>links will always be empty if a callback routine was provided when the the *HTML::LinkExtor* manpage was created.

EXAMPLE

This is an example showing how you can extract links as a document is received using LWP:

```
use LWP::UserAgent;
use HTML::LinkExtor;
use URI::URL;

$url = "http://www.sn.no/";  # for instance
$ua = new LWP::UserAgent;

# Set up a callback that collect image links
my @imgs = ();
sub callback {
   my($tag, %attr) = @_;
   return if $tag ne 'img';  # we only look closer at <img ...>
   push(@imgs, values %attr);
}
```

```
# Make the parser.  Unfortunately, we don't know the base yet (it might
# be diffent from $url)
$p = HTML::LinkExtor->new(\&callback);

# Request document and parse it as it arrives
$res = $ua->request(HTTP::Request->new(GET => $url), sub {$p->parse($_[0])});

# Expand all image URLs to absolute ones
my $base = $res->base;
@imgs = map { $_ = url($_, $base)->abs; } @imgs;

# Print them out
print join("\n", @imgs), "\n";
```

SEE ALSO

the *HTML::Parser* manpage

AUTHOR

Gisle Aas <aas@sn.no>

NAME

parse_html – Parse HTML text

parse_htmlfile – Parse HTML text from file

SYNOPSIS

```
use HTML::Parse;
$h = parse_htmlfile("test.html");
print $h->dump;
$h = parse_html("<p>Some more <i>italic</i> text", $h);
$h->delete;

print parse_htmlfile("index.html")->as_HTML;  # tidy up markup in a file
```

DESCRIPTION

Disclaimer: This module is provided for backwards compatibility with earlier versions of this library. New code will probably prefer to use the HTML::Parser and HTML::TreeBuilder modules directly.

The `HTML::Parse` module provides functions to parse HTML documents. There are two functions exported by this module:

parse_html($html, [$obj])
: This function is really just a synonym for `$obj`–>*parse* ($html) and `$obj` is assumed to be a subclass of `HTML::Parser`. Refer to the *HTML::Parser* manpage for more documentation.

  The `$obj` will default to a internally created `HTML::TreeBuilder` object. This class implements a parser that builds (and is) a HTML syntax tree with HTML::Element objects as nodes.

  The return value from *parse_html()* is `$obj`.

parse_htmlfile($file, [$obj])
: Same as *parse_html()*, but obtains HTML text from the named file.

  Returns `undef` if the file could not be opened, or `$obj` otherwise.

When a `HTML::TreeBuilder` object is created, the following variables control how parsing takes place:

$HTML::Parse::IMPLICIT_TAGS
: Setting this variable to true will instruct the parser to try to deduce implicit elements and implicit end tags. If this variable is false you get a parse tree that just reflects the text as it stands. Might be useful for quick & dirty parsing. Default is true.

  Implicit elements have the *implicit()* attribute set.

$HTML::Parse::IGNORE_UNKNOWN
: This variable contols whether unknow tags should be represented as elements in the parse tree. Default is true.

$HTML::Parse::IGNORE_TEXT
: Do not represent the text content of elements. This saves space if all you want is to examine the structure of the document. Default is false.

$HTML::Parse::WARN
: Call *warn()* with an apropriate message for syntax errors. Default is false.

SEE ALSO

the *HTML::Parser* manpage, the *HTML::TreeBuilder* manpage, the *HTML::Element* manpage

COPYRIGHT

AUTHOR

Gisle Aas <aas@sn.no>

NAME

HTML::Parser – SGML parser class

SYNOPSIS

```
require HTML::Parser;
$p = HTML::Parser->new;  # should really a be subclass
$p->parse($chunk1);
$p->parse($chunk2);
#...
$p->eof;                 # signal end of document

# Parse directly from file
$p->parse_file("foo.html");
# or
open(F, "foo.html") || die;
$p->parse_file(\*F);
```

DESCRIPTION

The `HTML::Parser` will tokenize a HTML document when the `$p`–>*parse()* method is called. The document to parse can be supplied in arbitrary chunks. Call `$p`–>*eof()* the end of the document to flush any remaining text. The return value from *parse()* is a reference to the parser object.

The `$p`–>*parse_file()* method can be called to parse text from a file. The argument can be a filename or an already opened file handle. The return value from *parse_file()* is a reference to the parser object.

In order to make the parser do anything interesting, you must make a subclass where you override one or more of the following methods as appropriate:

$self->declaration($decl)
: This method is called when a *markup declaration* has been recognized. For typical HTML documents, the only declaration you are likely to find is <!DOCTYPE ...>. The initial "<!" and ending ">" is not part of the string passed as argument. Comments are removed and entities have **not** been expanded yet.

$self->start($tag, $attr, $attrseq, $origtext)
: This method is called when a complete start tag has been recognized. The first argument is the tag name (in lower case) and the second argument is a reference to a hash that contain all attributes found within the start tag. The attribute keys are converted to lower case. Entities found in the attribute values are already expanded. The third argument is a reference to an array with the lower case attribute keys in the original order. The fourth argument is the original HTML text.

$self->end($tag)
: This method is called when an end tag has been recognized. The argument is the lower case tag name.

$self->text($text)
: This method is called when plain text in the document is recognized. The text is passed on unmodified and might contain multiple lines. Note that for efficiency reasons entities in the text are **not** expanded. You should call HTML::Entities::*decode* ($text) before you process the text any further.

$self->comment($comment)
: This method is called as comments are recognized. The leading and trailing "--" sequences have been stripped off the comment text.

The default implementation of these methods does nothing, *i.e.,* the tokens are just ignored.

There is really nothing in the basic parser that is HTML specific, so it is likely that the parser can parse many kinds of SGML documents, but SGML has many obscure features (not implemented by this module) that prevent us from renaming this module as `SGML::Parse`.

BUGS

You can instruct the parser to parse comments the way Netscape does it by calling the *netscape_buggy_comment()* method with a TRUE argument. This means that comments will always be terminated by the first occurence of "—>".

SEE ALSO

the *HTML::TreeBuilder* manpage, the *HTML::HeadParser* manpage, the *HTML::Entities* manpage

COPYRIGHT

AUTHOR

Gisle Aas <aas@sn.no>

NAME

HTML::TreeBuilder − Parser that builds a HTML syntax tree

SYNOPSIS

```
$h = new HTML::TreeBuilder;
$h->parse($document);
#...

print $h->as_HTML;  # or any other HTML::Element method
```

DESCRIPTION

This is a parser that builds (and actually itself is) a HTML syntax tree.

Objects of this class inherit the methods of both `HTML::Parser` and `HTML::Element`. After parsing has taken place it can be regarded as the syntax tree itself.

The following method all control how parsing takes place. You can set the attributes by passing a TRUE or FALSE value as argument.

$p->implicit_tags

Setting this attribute to true will instruct the parser to try to deduce implicit elements and implicit end tags. If it is false you get a parse tree that just reflects the text as it stands. Might be useful for quick & dirty parsing. Default is true.

Implicit elements have the *implicit()* attribute set.

$p->ignore_unknown

This attribute controls whether unknown tags should be represented as elements in the parse tree. Default is true.

$p->ignore_text

Do not represent the text content of elements. This saves space if all you want is to examine the structure of the document. Default is false.

$p->warn

Call *warn()* with an appropriate message for syntax errors. Default is false.

SEE ALSO

the *HTML::Parser* manpage, the *HTML::Element* manpage

COPYRIGHT

AUTHOR

Gisle Aas <aas@sn.no>

NAME

HTTP::Daemon − a simple http server class

SYNOPSIS

```
use HTTP::Daemon;
use HTTP::Status;

$d = new HTTP::Daemon;
print "Please contact me at: <URL:", $d->url, ">\n";
while ($c = $d->accept) {
    $r = $c->get_request;
    if ($r) {
        if ($r->method eq 'GET' and $r->url->path eq "/xyzzy") {
            # this is *not* recommened practice
            $c->send_file_response("/etc/passwd");
        } else {
            $c->send_error(RC_FORBIDDEN)
        }
    }
    $c = undef;  # close connection
}
```

DESCRIPTION

Instances of the *HTTP::Daemon* class are HTTP/1.1 servers that listens on a socket for incoming requests. The *HTTP::Daemon* is a sub-class of *IO::Socket::INET*, so you can do socket operations directly on it.

The *accept()* method will return when a connection from a client is available. The returned value will be a reference to a object of the *HTTP::Daemon::ClientConn* class which is another *IO::Socket::INET* subclass. Calling the *get_request()* method on this object will read data from the client and return an *HTTP::Request* object reference.

This HTTP daemon does not *fork*(2) for you. Your application, i.e. the user of the *HTTP::Daemon* is reponsible for forking if that is desirable. Also note that the user is responsible for generating responses that conforms to the HTTP/1.1 protocol. The *HTTP::Daemon::ClientConn* provide some methods that make this easier.

METHODS

The following is a list of methods that are new (or enhanced) relative to the *IO::Socket::INET* base class.

$d = new HTTP::Daemon

The object constructor takes the same parameters as the *IO::Socket::INET* constructor. It can also be called without specifying any parameters. The daemon will then set up a listen queue of 5 connections and allocate some random port number. A server that want to bind to some specific address on the standard HTTP port will be constructed like this:

```
$d = new HTTP::Daemon
      LocalAddr => 'www.someplace.com',
      LocalPort => 80;
```

$c = $d->accept

Same as *IO::Socket::accept* but will return an *HTTP::Daemon::ClientConn* reference. It will return undef if you have specified a timeout and no connection is made within that time.

$d->url

Returns a URL string that can be used to access the server root.

$d->product_tokens
: Returns the name that this server will use to identify itself. This is the string that is sent with the *Server* response header.

The *HTTP::Daemon::ClientConn* is also a *IO::Socket::INET* subclass. Instances of this class are returned by the *accept()* method of the *HTTP::Daemon*. The following additional methods are provided:

$c->get_request
: Will read data from the client and turn it into a *HTTP::Request* object which is then returned. Will return undef if reading of the request failed. If it fails, then the *HTTP::Daemon::ClientConn* object ($c) should be discarded.

: The `$c->`get_request method support HTTP/1.1 content bodies, including *chunked* transfer encoding with footer and *multipart/** types.

$c->antique_client
: Returns TRUE if the client speaks the HTTP/0.9 protocol, i.e. no status code or headers should be returned.

$c->send_status_line([$code, [$mess, [$proto]]])
: Sends the status line back to the client.

$c->send_basic_header([$code, [$mess, [$proto]]])
: Sends the status line and the ‘‘Date:’’ and ‘‘Server:’’ headers back to the client.

$c->send_response([$res])
: Takes a *HTTP::Response* object as parameter and send it back to the client as the response.

$c->send_redirect($loc, [$code, [$entity_body]])
: Sends a redirect response back to the client. The location ($loc) can be an absolute or a relative URL. The `$code` must be one the redirect status codes, and it defaults to ‘‘301 Moved Permanently’’

$c->send_error([$code, [$error_message]])
: Send an error response back to the client. If the `$code` is missing a ‘‘Bad Request’’ error is reported. The `$error_message` is a string that is incorporated in the body of the HTML entity body.

$c->send_file_response($filename)
: Send back a response with the specified `$filename` as content. If the file happen to be a directory we will generate a HTML index for it.

$c->send_file($fd);
: Copies the file back to the client. The file can be a string (which will be interpreted as a filename) or a reference to a glob.

$c->daemon
: Return a reference to the corresponding *HTTP::Daemon* object.

SEE ALSO

the *IO::Socket* manpage, the *Apache* manpage

COPYRIGHT

NAME

time2str, str2time – date conversion routines

SYNOPSIS

```
use HTTP::Date;

$stringGMT = time2str(time);   # Format as GMT ASCII time
$time = str2time($stringGMT);  # convert ASCII date to machine time
```

DESCRIPTION

This module provides two functions that deal with the HTTP date format.

time2str ([$time])

The *time2str()* function converts a machine time (seconds since epoch) to a string. If the function is called without an argument, it will use the current time.

The string returned is in the format defined by the HTTP/1.0 specification. This is a fixed length subset of the format defined by RFC 1123, represented in Universal Time (GMT). An example of this format is:

```
Thu, 03 Feb 1994 17:09:00 GMT
```

str2time($str [, `$zone`])

The *str2time()* function converts a string to machine time. It returns `undef` if the format is unrecognized, or the year is not between 1970 and 2038. The function is able to parse the following formats:

```
"Wed, 09 Feb 1994 22:23:32 GMT"       -- HTTP format
"Thu Feb  3 17:03:55 GMT 1994"        -- ctime(3) format
"Thu Feb  3 00:00:00 1994",           -- ANSI C asctime() format
"Tuesday, 08-Feb-94 14:15:29 GMT"     -- old rfc850 HTTP format
"Tuesday, 08-Feb-1994 14:15:29 GMT"   -- broken rfc850 HTTP format

"03/Feb/1994:17:03:55 -0700"   -- common logfile format
"09 Feb 1994 22:23:32 GMT"     -- HTTP format (no weekday)
"08-Feb-94 14:15:29 GMT"       -- rfc850 format (no weekday)
"08-Feb-1994 14:15:29 GMT"     -- broken rfc850 format (no weekday)

"1994-02-03 14:15:29 -0100"    -- ISO 8601 format
"1994-02-03 14:15:29"          -- zone is optional
"1994-02-03"                   -- only date
"1994-02-03T14:15:29"          -- Use T as separator
"19940203T141529Z"             -- ISO 8601 compact format
"19940203"                     -- only date

"08-Feb-94"         -- old rfc850 HTTP format    (no weekday, no time)
"08-Feb-1994"       -- broken rfc850 HTTP format (no weekday, no time)
"09 Feb 1994"       -- proposed new HTTP format  (no weekday, no time)
"03/Feb/1994"       -- common logfile format     (no time, no offset)

"Feb  3  1994"      -- Unix 'ls -l' format
"Feb  3 17:03"      -- Unix 'ls -l' format

"11-15-96  03:52PM" -- Windows 'dir' format
```

The parser ignores leading and trailing whitespace. It also allow the seconds to be missing and the month to be numerical in most formats.

The *str2time()* function takes an optional second argument that specifies the default time zone to use when converting the date. This zone specification should be numerical (like ‘‘−0800’’ or ‘‘+0100") or ‘‘GMT’’. This parameter is ignored if the zone is specified in the date string itself. It this parameter is missing, and the date string format does not contain any zone specification then the local time zone is assumed.

If the year is missing, then we assume that the date is the first matching date *before* current time.

BUGS

Non-numerical time zones (like MET, PST) are all treated like GMT. Do not use them. HTTP does not use them.

The *str2time()* function has been told how to parse far too many formats. This makes the module name misleading. To be sure it is really misleading you can also import the *time2iso()* and *time2isoz()* functions. They work like *time2str()* but produce ISO−8601 formated strings (YYYY−MM−DD hh:mm:ss).

COPYRIGHT

NAME

HTTP::Headers – Class encapsulating HTTP Message headers

SYNOPSIS

```
require HTTP::Headers;
$request = new HTTP::Headers;
```

DESCRIPTION

The `HTTP::Headers` class encapsulates HTTP–style message headers. The headers consist of attribute-value pairs, which may be repeated, and which are printed in a particular order.

Instances of this class are usually created as member variables of the `HTTP::Request` and `HTTP::Response` classes, internal to the library.

METHODS

`$h` = new HTTP::Headers

Constructs a new `HTTP::Headers` object. You might pass some initial attribute-value pairs as parameters to the constructor. *E.g.*:

```
$h = new HTTP::Headers
    Date         => 'Thu, 03 Feb 1994 00:00:00 GMT',
    Content_Type => 'text/html; version=3.2',
    Content_Base => 'http://www.sn.no/';
```

`$h`–>header($field [=> `$val`],...)

Get or set the value of a header. The header field name is not case sensitive. To make the life easier for perl users who wants to avoid quoting before the => operator, you can use '_' as a synonym for '–' in header names.

The value argument may be a scalar or a reference to a list of scalars. If the value argument is not defined, then the header is not modified.

The *header()* method accepts multiple ($field => `$value`) pairs.

The list of previous values for the last `$field` is returned. Only the first header value is returned in scalar context.

```
$header->header(MIME_Version => '1.0',
                User_Agent   => 'My-Web-Client/0.01');
$header->header(Accept => "text/html, text/plain, image/*");
$header->header(Accept => [qw(text/html text/plain image/*)]);
@accepts = $header->header('Accept');
```

`$h`–>*scan* (\&doit)

Apply a subroutine to each header in turn. The callback routine is called with two parameters; the name of the field and a single value. If the header has more than one value, then the routine is called once for each value. The field name passed to the callback routine has case as suggested by HTTP Spec, and the headers will be visited in the recommended "Good Practice" order.

`$h`–>*as_string* ([$endl])

Return the header fields as a formatted MIME header. Since it internally uses the `scan()` method to build the string, the result will use case as suggested by HTTP Spec, and it will follow recommended "Good Practice" of ordering the header fieds. Long header values are not folded.

The optional parameter specifies the line ending sequence to use. The default is `"\n"`. Embedded "\n" characters in the header will be substitued with this line ending sequence.

$h->push_header($field, $val)

Add a new field value of the specified header. The header field name is not case sensitive. The field need not already have a value. Previous values for the same field are retained. The argument may be a scalar or a reference to a list of scalars.

```
$header->push_header(Accept => 'image/jpeg');
```

$h->remove_header($field,...)

This function removes the headers with the specified names.

$h->clone

Returns a copy of this HTTP::Headers object.

CONVENIENCE METHODS

The most frequently used headers can also be accessed through the following convenience methods. These methods can both be used to read and to set the value of a header. The header value is set if you pass an argument to the method. The old header value is always returned.

Methods that deal with dates/times always convert their value to system time (seconds since Jan 1, 1970) and they also expect this kind of value when the header value is set.

$h->date

This header represents the date and time at which the message was originated. *E.g.*:

```
$h->date(time);  # set current date
```

$h->expires

This header gives the date and time after which the entity should be considered stale.

$h->if_modified_since

This header is used to make a request conditional. If the requested resource has not been modified since the time specified in this field, then the server will return a `"304 Not Modified"` response instead of the document itself.

$h->last_modified

This header indicates the date and time at which the resource was last modified. *E.g.*:

```
# check if document is more than 1 hour old
if ($h->last_modified < time - 60*60) {
      ...
}
```

$h->content_type

The Content-Type header field indicates the media type of the message content. *E.g.*:

```
$h->content_type('text/html');
```

The value returned will be converted to lower case, and potential parameters will be chopped off and returned as a separate value if in an array context. This makes it safe to do the following:

```
if ($h->content_type eq 'text/html') {
   # we enter this place even if the real header value happens to
   # be 'TEXT/HTML; version=3.0'
   ...
}
```

$h->content_encoding

The Content-Encoding header field is used as a modifier to the media type. When present, its value indicates what additional encoding mechanism has been applied to the resource.

$h->content_length

A decimal number indicating the size in bytes of the message content.

$h->title

The title of the document. In libwww-perl this header will be initialized automatically from the <TITLE>...</TITLE> element of HTML documents. *This header is no longer part of the HTTP standard.*

$h->user_agent

This header field is used in request messages and contains information about the user agent originating the request. *E.g.*:

```
$h->user_agent('Mozilla/1.2');
```

$h->server

The server header field contains information about the software being used by the originating server program handling the request.

$h->from

This header should contain an Internet e-mail address for the human user who controls the requesting user agent. The address should be machine-usable, as defined by RFC822. E.g.:

```
$h->from('Gisle Aas <aas@sn.no>');
```

$h->referer

Used to specify the address (URI) of the document from which the requested resouce address was obtained.

$h->www_authenticate

This header must be included as part of a "401 Unauthorized" response. The field value consist of a challenge that indicates the authentication scheme and parameters applicable to the requested URI.

$h->authorization

A user agent that wishes to authenticate itself with a server, may do so by including this header.

$h->authorization_basic

This method is used to get or set an authorization header that use the "Basic Authentication Scheme". In array context it will return two values; the user name and the password. In scalar context it will return *"uname:password"* as a single string value.

When used to set the header value, it expects two arguments. *E.g.*:

```
$h->authorization_basic($uname, $password);
```

NAME

HTTP::Message – Class encapsulating HTTP messages

DESCRIPTION

A `HTTP::Message` object contains some headers and a content (body). The class is abstract, i.e. it only used as a base class for `HTTP::Request` and `HTTP::Response` and should never instantiated as itself.

METHODS

`$mess` = new HTTP::Message;

This is the object constructor. It should only be called internally by this library. External code should construct `HTTP::Request` or `HTTP::Response` objects.

`$mess`–>*clone()*

Returns a copy of the object.

`$mess`–>*protocol* ([$proto])

Sets the HTTP protocol used for the message. The *protocol()* is a string like "HTTP/1.0" or "HTTP/1.1".

`$mess`–>*content* ([$content])

The *content()* method sets the content if an argument is given. If no argument is given the content is not touched. In either case the previous content is returned.

`$mess`–>*add_content* ($data)

The *add_content()* methods appends more data to the end of the previous content.

`$mess`–>content_ref

The *content_ref()* method will return a reference to content string. It can be more efficient to access the content this way if the content is huge, and it can be used for direct manipulation of the content, for instance:

```
${$res->content_ref} =~ s/\bfoo\b/bar/g;
```

HEADER METHODS

All unknown `HTTP::Message` methods are delegated to the `HTTP::Headers` object that is part of every message. This allows convenient access to these methods. Refer to the *HTTP::Headers* manpage for details of these methods:

```
$mess->header($field => $val);
$mess->scan(&doit);
$mess->push_header($field => $val);
$mess->remove_header($field);
```

```
$mess->date;
$mess->expires;
$mess->if_modified_since;
$mess->last_modified;
$mess->content_type;
$mess->content_encoding;
$mess->content_length;
$mess->title;
$mess->user_agent;
$mess->server;
$mess->from;
$mess->referer;
$mess->www_authenticate;
$mess->authorization;
$mess->authorization_basic;
```

`$mess->`*headers_as_string* ([$endl])

Call the HTTP::Headers->*as_string()* method for the headers in the message.

NAME

choose – choose a variant of a document to serve (HTTP content negotiation)

SYNOPSIS

```
use HTTP::Negotiate;

#  ID       QS     Content-Type   Encoding Char-Set        Lang   Size
$variants =
 [['var1',  1.000, 'text/html',   undef,   'iso-8859-1',   'en',   3000],
  ['var2',  0.950, 'text/plain',  'gzip',  'us-ascii',     'no',    400],
  ['var3',  0.3,   'image/gif',   undef,   undef,          undef, 43555],
 ];

@prefered = choose($variants, $request_headers);
$the_one  = choose($variants);
```

DESCRIPTION

This module provide a complete implementation of the HTTP content negotiation algorithm specified in *draft-ietf-http-v11-spec-00.ps* chapter 12. Content negotiation allows for the selection of a preferred content representation based upon attributes of the negotiable variants and the value of the various Accept* header fields in the request.

The variants are ordered by preference by calling the function *choose()*.

The first parameter is a description of the variants that we can choose among. The variants are described by a reference to an array. Each element in this array is an array with the values [$id, `$qs`, `$content_type`, `$content_encoding`, `$charset`, `$content_language`, `$content_length`]. The meaning of these values are described below. The `$content_encoding` and `$content_language` can be either a single scalar value or an array reference if there are several values.

The second optional parameter is a reference to the request headers. This is used to look for "Accept*" headers. You can pass a reference to either a HTTP::Request or a HTTP::Headers object. If this parameter is missing, then the accept specification is initialized from the CGI environment variables HTTP_ACCEPT, HTTP_ACCEPT_CHARSET, HTTP_ACCEPT_ENCODING and HTTP_ACCEPT_LANGUAGE.

In array context, *choose()* returns a list of variant identifier/calculated quality pairs. The values are sorted by quality, highest quality first. If the calculated quality is the same for two variants, then they are sorted by size (smallest first). *E.g.*:

```
(['var1' => 1], ['var2', 0.3], ['var3' => 0]);
```

Note that also zero quality variants are included in the return list even if these should never be served to the client.

In scalar context, it returns the identifier of the variant with the highest score or `undef` if none have non-zero quality.

If the `$HTTP::Negotiate::DEBUG` variable is set to TRUE, then a lot of noise is generated on STDOUT during evaluation of *choose()*.

VARIANTS

A variant is described by a list of the following values. If the attribute does not make sense or is unknown for a variant, then use `undef` instead.

identifier

This is just some string that you use as a name for the variant. The identifier of the preferred variant is returned by *choose()*.

qs This is a number between 0.000 and 1.000 that describes the "source quality". This is what *draft-ietf-http-v11-spec-00.ps* says about this value:

Source quality is measured by the content provider as representing the amount of degradation from the original source. For example, a picture in JPEG form would have a lower qs when translated to the XBM format, and much lower qs when translated to an ASCII−art representation. Note, however, that this is a function of the source − an original piece of ASCII−art may degrade in quality if it is captured in JPEG form. The qs values should be assigned to each variant by the content provider; if no qs value has been assigned, the default is generally "qs=1".

content-type
This is the media type of the variant. The media type does not include a charset attribute, but might contain other parameters. Examples are:

```
text/html
text/html;version=2.0
text/plain
image/gif
image/jpg
```

content-encoding
This is one or more content encodings that has been applied to the variant. The content encoding is generally used as a modifier to the content media type. The most common content encodings are:

```
gzip
compress
```

content-charset
This is the character set used when the variant contains textual content. The charset value should generally be `undef` or one of these:

```
us-ascii
iso-8859-1 ... iso-8859-9
iso-2022-jp
iso-2022-jp-2
iso-2022-kr
unicode-1-1
unicode-1-1-utf-7
unicode-1-1-utf-8
```

content-language
This describes one or more languages that are used in the variant. Language is described like this in *draft-ietf-http-v11-spec-00.ps*: A language is in this context a natural language spoken, written, or otherwise conveyed by human beings for communication of information to other human beings. Computer languages are explicitly excluded.

The language tags are the same as those defined by RFC−1766. Examples are:

```
no              Norwegian
en              International English
en-US           US English
en-cockney
```

content-length
This is the number of bytes used to represent the content.

ACCEPT HEADERS

The following Accept* headers can be used for describing content preferences in a request (This description is an edited extract from *draft-ietf-http-v11-spec-00.ps*):

Accept

This header can be used to indicate a list of media ranges which are acceptable as a reponse to the request. The "*" character is used to group media types into ranges, with "*/*" indicating all media types and "type/*" indicating all subtypes of that type.

The parameter q is used to indicate the quality factor, which represents the user's preference for that range of media types. The parameter mbx gives the maximum acceptable size of the response content. The default values are: q=1 and mbx=infinity. If no Accept header is present, then the client accepts all media types with q=1.

For example:

```
Accept: audio/*;q=0.2;mbx=200000, audio/basic
```

would mean: "I prefer audio/basic (of any size), but send me any audio type if it is the best available after an 80% mark-down in quality and its size is less than 200000 bytes"

Accept-Charset

Used to indicate what character sets are acceptable for the response. The "us-ascii" character set is assumed to be acceptable for all user agents. If no Accept-Charset field is given, the default is that any charset is acceptable. Example:

```
Accept-Charset: iso-8859-1, unicode-1-1
```

Accept-Encoding

Restricts the Content-Encoding values which are acceptable in the response. If no Accept-Encoding field is present, the server may assume that the client will accept any content encoding. An empty Accept-Encoding means that no content encoding is acceptable. Example:

```
Accept-Encoding: compress, gzip
```

Accept-Language

This field is similar to Accept, but restrict the set of natural languages that are preferred as a response. Each language may be given an associated quality value which represents an estimate of the user's comprehension of that language. For example:

```
Accept-Language: no, en-gb;q=0.8, de;q=0.55
```

would mean: "I prefer Norwegian, but will accept British English (with 80% comprehension) or German (with 55% comprehension).

COPYRIGHT

AUTHOR

Gisle Aas <aas@sn.no>

NAME

HTTP::Request – Class encapsulating HTTP Requests

SYNOPSIS

```
require HTTP::Request;
$request = new HTTP::Request 'GET', 'http://www.oslonett.no/';
```

DESCRIPTION

`HTTP::Request` is a class encapsulating HTTP style requests, consisting of a request line, a MIME header, and optional content. Note that the LWP library also uses this HTTP style requests for non-HTTP protocols.

Instances of this class are usually passed to the `request()` method of an `LWP::UserAgent` object:

```
$ua = new LWP::UserAgent;
$request = new HTTP::Request 'http://www.oslonett.no/';
$response = $ua->request($request);
```

METHODS

`HTTP::Request` is a subclass of `HTTP::Message` and therefore inherits its methods. The inherited methods are *header()*, *push_header()*, *remove_header()*, *headers_as_string()* and *content()*. See the *HTTP::Message* manpage for details.

`$r` = new HTTP::Request `$method`, `$url`, [$header, [$content]]

Constructs a new `HTTP::Request` object describing a request on the object `$url` using method `$method`. The `$url` argument can be either a string, or a reference to a `URI::URL` object. The `$header` argument should be a reference to a HTTP::Headers object.

```
$request = new HTTP::Request 'GET', 'http://www.oslonett.no/';
```

`$r`–>*method* ([$val])

`$r`–>*url* ([$val])

These methods provide public access to the member variables containing respectively the method of the request and the URL object of the request.

If an argument is given the member variable is given that as its new value. If no argument is given the value is not touched. In either case the previous value is returned.

The *url()* method accept both a reference to a URI::URL object and a string as its argument. If a string is given, then it should be parseable as an absolute URL.

`$r`–>*as_string()*

Method returning a textual representation of the request. Mainly useful for debugging purposes. It takes no arguments.

NAME

HTTP::Response − Class encapsulating HTTP Responses

SYNOPSIS

```
require HTTP::Response;
```

DESCRIPTION

The `HTTP::Response` class encapsulate HTTP style responses. A response consist of a response line, some headers, and a (potential empty) content. Note that the LWP library will use HTTP style responses also for non-HTTP protocol schemes.

Instances of this class are usually created and returned by the `request()` method of an `LWP::UserAgent` object:

```
...
$response = $ua->request($request)
if ($response->is_success) {
    print $response->content;
} else {
    print $response->error_as_HTML;
}
```

METHODS

`HTTP::Response` is a subclass of `HTTP::Message` and therefore inherits its methods. The inherited methods are *header()*, *push_header()*, *remove_header()*, *headers_as_string()*, and *content()*. The header convenience methods are also available. See the *HTTP::Message* manpage for details.

`$r` = new HTTP::Response ($rc, [$msg, [$header, [$content]]])

Constructs a new `HTTP::Response` object describing a response with response code `$rc` and optional message `$msg`.

`$r`−>*code* ([$code])

`$r`−>*message* ([$message])

`$r`−>*request* ([$request])

`$r`−>*previous* ([$previousResponse])

These methods provide public access to the member variables. The first two containing respectively the response code and the message of the response.

The request attribute is a reference the request that gave this response. It does not have to be the same request as passed to the `$ua`−>*request()* method, because there might have been redirects and authorization retries in between.

The previous attribute is used to link together chains of responses. You get chains of responses if the first response is redirect or unauthorized.

`$r`−>base

Returns the base URL for this response. The return value will be a reference to a URI::URL object.

The base URL is obtained from one the following sources (in priority order):

1. Embedded in the document content, for instance <BASE HREF=’’..."> in HTML documents.

2. A “Content-Base:” or a “Content-Location:” header in the response.

   For backwards compatability with older HTTP implementations we will also look for the “Base:” header.

3. The URL used to request this response. This might not be the original URL that was passed to `$ua`–>*request()* method, because we might have received some redirect responses first.

When the LWP protocol modules produce the HTTP::Response object, then any base URL embedded in the document (step 1) will already have initialized the “Content-Base:” header. This means that this method only perform the last 2 steps (the content is not always available either).

`$r`–>*as_string()*

Method returning a textual representation of the request. Mainly useful for debugging purposes. It takes no arguments.

`$r`–>is_info

`$r`–>is_success

`$r`–>is_redirect

`$r`–>is_error

These methods indicate if the response was informational, sucessful, a redirection, or an error.

`$r`–>*error_as_HTML()*

Return a string containing a complete HTML document indicating what error occurred. This method should only be called when `$r`–>is_error is TRUE.

`$r`–>current_age

This function will calculate the “current age” of the response as specified by <draft-ietf-http-v11-spec-07> section 13.2.3. The age of a response is the time since it was sent by the origin server. The returned value is a number representing the age in seconds.

`$r`–>freshness_lifetime

This function will calculate the “freshness lifetime” of the response as specified by <draft-ietf-http-v11-spec-07> section 13.2.4. The “freshness lifetime” is the length of time between the generation of a response and its expiration time. The returned value is a number representing the freshness lifetime in seconds.

If the response does not contain an “Expires” or a “Cache-Control” header, then this function will apply some simple heuristic based on ‘Last-Modified’ to determine a suitable lifetime.

`$r`–>is_fresh

Returns TRUE if the response is fresh, based on the values of *freshness_lifetime()* and *current_age()*. If the response is not longer fresh, then it has to be refetched or revalidated by the origin server.

`$r`–>fresh_until

Returns the time when this entiy is no longer fresh.

NAME

HTTP::Status − HTTP Status code processing

SYNOPSIS

```
use HTTP::Status;

if ($rc != RC_OK) {
    print status_message($rc), "\n";
}

if (is_success($rc)) { ... }
if (is_error($rc)) { ... }
if (is_redirect($rc)) { ... }
```

DESCRIPTION

HTTP::Status is a library of routines for defining and classification of HTTP status codes for libwww-perl. Status codes are used to encode the overall outcome of a HTTP response message. Codes correspond to those defined in <draft-ietf-http-v11-spec-06>.

The following functions can be used as mnemonic status code names:

```
RC_CONTINUE                           (100)
RC_SWITCHING_PROTOCOLS                (101)

RC_OK                                 (200)
RC_CREATED                            (201)
RC_ACCEPTED                           (202)
RC_NON_AUTHORITATIVE_INFORMATION      (203)
RC_NO_CONTENT                         (204)
RC_RESET_CONTENT                      (205)
RC_PARTIAL_CONTENT                    (206)

RC_MULTIPLE_CHOICES                   (300)
RC_MOVED_PERMANENTLY                  (301)
RC_MOVED_TEMPORARILY                  (302)
RC_SEE_OTHER                          (303)
RC_NOT_MODIFIED                       (304)
RC_USE_PROXY                          (305)

RC_BAD_REQUEST                        (400)
RC_UNAUTHORIZED                       (401)
RC_PAYMENT_REQUIRED                   (402)
RC_FORBIDDEN                          (403)
RC_NOT_FOUND                          (404)
RC_METHOD_NOT_ALLOWED                 (405)
RC_NOT_ACCEPTABLE                     (406)
RC_PROXY_AUTHENTICATION_REQUIRED      (407)
RC_REQUEST_TIMEOUT                    (408)
RC_CONFLICT                           (409)
RC_GONE                               (410)
RC_LENGTH_REQUIRED                    (411)
RC_PRECONDITION_FAILED                (412)
RC_REQUEST_ENTITY_TOO_LARGE           (413)
RC_REQUEST_URI_TOO_LARGE              (414)
RC_UNSUPPORTED_MEDIA_TYPE             (415)
```

```
RC_INTERNAL_SERVER_ERROR             (500)
RC_NOT_IMPLEMENTED                   (501)
RC_BAD_GATEWAY                       (502)
RC_SERVICE_UNAVAILABLE               (503)
RC_GATEWAY_TIMEOUT                   (504)
RC_HTTP_VERSION_NOT_SUPPORTED        (505)
```

The *status_message()* function will translate status codes to human readable strings.

The *is_info()*, *is_success()*, *is_redirect()*, and *is_error()* functions will return a TRUE value if the status code passed as argument is informational, indicates success, and error, or a redirect respectively.

status_message ($code)

Return user friendly error message for status code `$code`

is_info ($code)

Return TRUE if `$code` is an *Informational* status code.

is_success ($code)

Return TRUE if `$code` is a *Successful* status code.

is_redirect ($code)

Return TRUE if `$code` is a *Redirection* status code.

is_error ($code)

Return TRUE if `$code` is an *Error* status code. The `$code` can be both a client error or a server error.

is_client_error ($code)

Return TRUE if `$code` is an *Client Error* status code.

is_server_error ($code)

Return TRUE if `$code` is an *Server Error* status code.

NAME

LWP::Debug − debug routines for the libwww-perl library

SYNOPSIS

```
use LWP::Debug qw(+ -conns);

# Used internally in the library
LWP::Debug::trace('send()');
LWP::Debug::debug('url ok');
LWP::Debug::conns("read $n bytes: $data");
```

DESCRIPTION

LWP::Debug provides tracing facilities. The *trace()*, *debug()* and *conns()* function are called within the library and they log information at increasing levels of detail. Which level of detail is actually printed is controlled with the `level()` function.

FUNCTIONS

level (...)

The `level()` function controls the level of detail being logged. Passing '+' or '−' indicates full and no logging respectively. Inidividual levels can switched on and of by passing the name of the level with a '+' or '−' prepended. The levels are:

```
trace   : trace function calls
debug   : print debug messages
conns   : show all data transfered over the connections
```

The LWP::Debug module provide a special *import()* method that allows you to pass the *level()* arguments with initial use statement. If a use argument start with '+' or '−' then it is passed to the level function, else the name is exported as usual. The following two statements are thus equivalent (if you ignore that the second pollutes your namespace):

```
use LWP::Debug qw(+);
use LWP::Debug qw(level); level('+');
```

trace ($msg)

The `trace()` function is used for tracing function calls. The package and calling subroutine name is printed along with the passed argument. This should be called at the start of every major function.

debug ($msg)

The `debug()` function is used for high-granularity reporting of state in functions.

conns ($msg)

The `conns()` function is used to show data being transferred over the connections. This may generate considerable output.

NAME

LWP::IO – Low level I/O capability

DESCRIPTION

LWP::IO::read($fd, `$data`, `$size`, `$offset`, `$timeout`)

LWP::IO::write($fd, `$data`, `$timeout`)

These routines provide low level I/O with timeout capability for the LWP library. These routines will only be installed if they are not already defined. This fact can be used by programs that need to override these functions. Just provide replacement functions before you require LWP. See also the *LWP::TkIO* manpage.

NAME

guess_media_type – guess media type for a file or a URL.

media_suffix – returns file extentions for a media type

DESCRIPTION

This module provides functions for handling of media (also known as MIME) types and encodings. The mapping from file extentions to media types is defined by the *media.types* file. If the *~/.media.types* file exist it is used as a replacement.

For backwards compatability we will also look for *~/.mime.types*.

FUNCTIONS

guess_media_type ($filename)

This function tries to guess media type and encoding for given file. In scalar context it returns only the content-type. In array context it returns an array consisting of content-type followed by any content-encodings applied.

The guess_media_type function also accepts a URI::URL object as argument.

If the type can not be deduced from looking at the file name only, then *guess_media_type()* will take a look at the actual file using the `-T` perl operator in order to determine if this is a text file (text/plain). If this does not work it will return *application/octet-stream* as the type.

media_suffix($type,...)

This function will return all suffixes that can be used to denote the specified media *type* (s). Wildcard types can be used. In scalar context it will return the first suffix found.

Examples:

```
@suffixes = media_suffix('image/*', 'audio/basic');
$suffix = media_suffix('text/html');
```

NAME

LWP::MemberMixin − Member access mixin class

DESCRIPTION

A mixin class to get methods that provide easy access to member variables in the %$self.

BUGS

Ideally there should be better Perl langauge support for this.

METHODS

_elem($elem [, `$val`])

Internal method to get/set the value of member variable `$elem`. If `$val` is defined it is used as the new value for the member variable. If it is undefined the current value is not touched. In both cases the previous value of the member variable is returned.

NAME

LWP::Protocol − Base class for LWP protocols

DESCRIPTION

This class is used a the base class for all protocol implementations supported by the LWP library.

When creating an instance of this class using `LWP::Protocol::create($url)`, and you get an initialised subclass appropriate for that access method. In other words, the *LWP::Protocol::create()* function calls the constructor for one of its subclasses.

All derived LWP::Protocol classes need to override the *request()* method which is used to service a request. The overridden method can make use of the *collect()* function to collect together chunks of data as it is received.

SEE ALSO

Inspect the *LWP/Protocol/file.pm* and *LWP/Protocol/http.pm* files for examples of usage.

METHODS AND FUNCTIONS

`$prot` = new HTTP::Protocol;

The LWP::Protocol constructor is inherited by subclasses. As this is a virtual base class this method should **not** be called directly.

`$prot` = LWP::Protocol::*create* ($url)

Create an object of the class implementing the protocol to handle the given scheme. This is a function, not a method. It is more an object factory than a constructor. This is the function user agents should use to access protocols.

`$class` = LWP::Protocol::implementor($scheme, [$class])

Get and/or set implementor class for a scheme. Returns '' if the specified scheme is not supported.

`$prot`−>*request* (...)

```
 $response = $protocol->request($request, $proxy, undef);
 $response = $protocol->request($request, $proxy, '/tmp/sss');
 $response = $protocol->request($request, $proxy, \&callback, 1024);
```

Dispactches a request over the protocol, and returns a response object. This method needs to be overridden in subclasses. Referer to the *LWP::UserAgent* manpage for description of the arguments.

`$prot`−>*timeout* ($seconds)

Get and set the timeout value in seconds

`$prot`−>*use_alarm* ($yesno)

Indicates if the library is allowed to use the core *alarm()* function to implement timeouts.

`$prot`−>*parse_head* ($yesno)

Should we initialize response headers from the <head> section of HTML documents.

`$prot`−>collect($arg, `$response`, `$collector`)

Called to collect the content of a request, and process it appropriately into a scalar, file, or by calling a callback. If `$arg` is undefined, then the content is stored within the `$response`. If `$arg` is a simple scalar, then `$arg` is interpreted as a file name and the content is written to this file. If `$arg` is a reference to a routine, then content is passed to this routine.

The `$collector` is a routine that will be called and which is reponsible for returning pieces (as ref to scalar) of the content to process. The `$collector` signals EOF by returning a reference to an empty sting.

The return value from *collect()* is the `$response` object reference.

Note: We will only use the callback or file argument if `$response->`*is_success()*. This avoids sendig content data for redirects and authentization responses to the callback which would be confusing.

`$prot->`collect_once($arg, `$response`, `$content`)

Can be called when the whole response content is available as `$content`. This will invoke *collect()* with a collector callback that returns a reference to `$content` the first time and an empty string the next.

NAME

LWP::RobotUA − A class for Web Robots

SYNOPSIS

```
require LWP::RobotUA;
$ua = new LWP::RobotUA 'my-robot/0.1', 'me@foo.com';
$ua->delay(10);  # be very nice, go slowly
...
# just use it just like a normal LWP::UserAgent
$res = $ua->request($req);
```

DESCRIPTION

This class implements a user agent that is suitable for robot applications. Robots should be nice to the servers they visit. They should consult the *robots.txt* file to ensure that they are welcomed and they should not send too frequent requests.

But, before you consider writing a robot take a look at <URL:http://info.webcrawler.com/mak/projects/robots/robots.html>.

When you use a *LWP::RobotUA* as your user agent, then you do not really have to think about these things yourself. Just send requests as you do when you are using a normal *LWP::UserAgent* and this special agent will make sure you are nice.

METHODS

The LWP::RobotUA is a sub-class of LWP::UserAgent and implements the same methods. The *use_alarm()* method also desides whether we will wait if a request is tried too early (if true), or will return an error response (if false).

In addition these methods are provided:

`$ua` = LWP::RobotUA−>new($agent_name, `$from`, [$rules])

Your robot's name and the mail address of the human responsible for the robot (i.e. you) is required by the constructor.

Optionally it allows you to specify the *WWW::RobotRules* object to use.

`$ua`−>*delay* ([$minutes])

Set the minimum delay between requests to the same server. The default is 1 minute.

`$ua`−>*rules* ([$rules])

Set/get which *WWW::RobotRules* object to use.

`$ua`−>*no_visits* ($netloc)

Returns the number of documents fetched from this server host.

`$ua`−>*host_wait* ($netloc)

Returns the number of seconds you must wait before you can make a new request to this host.

`$ua`−>as_string

Returns a text that describe the state of the UA. Mainly useful for debugging.

SEE ALSO

the *LWP::UserAgent* manpage, the *WWW::RobotRules* manpage

AUTHOR

Gisle Aas <aas@sn.no>

NAME

get, head, getprint, getstore, mirror – Procedural LWP interface

SYNOPSIS

```
perl -MLWP::Simple -e 'getprint "http://www.sn.no"'

use LWP::Simple;
$content = get("http://www.sn.no/")
if (mirror("http://www.sn.no/", "foo") == RC_NOT_MODIFIED) {
    ...
}
if (is_success(getprint("http://www.sn.no/"))) {
    ...
}
```

DESCRIPTION

This interface is intended for those who want a simplified view of the libwww-perl library. This interface should also be suitable for one-liners. If you need more control or access to the header fields in the requests sent and responses received you should use the full OO interface provided by the LWP::UserAgent module.

This following functions are provided (and exported) by this module:

get($url)
: This function will get the document identified by the given URL. The *get()* function will return the document if successful or 'undef' if it fails. The `$url` argument can be either a simple string or a reference to a URI::URL object.

  You will not be able to examine the response code or response headers (like *Content-Type*) when you are accessing the web using this function. If you need this you should use the full OO interface.

head($url)
: Get document headers. Returns the following values if successful: ($content_type, `$document_length`, `$modified_time`, `$expires`, `$server`)

  Returns an empty list if it fails.

getprint($url)
: Get and print a document identified by a URL. The document is printet on STDOUT. The error message (formatted as HTML) is printed on STDERR if the request fails. The return value is the HTTP response code.

getstore($url, $file)
: Gets a document identified by a URL and stores it in the file. The return value is the HTTP response code.

mirror($url, $file)
: Get and store a document identified by a URL, using *If-modified-since*, and checking of the *Content-Length*. Returns the HTTP response code.

This module also exports the HTTP::Status constants and procedures. These can be used when you check the response code from *getprint()*, *getstore()* and *mirror()*. The constants are:

```
RC_CONTINUE
RC_SWITCHING_PROTOCOLS
RC_OK
RC_CREATED
RC_ACCEPTED
RC_NON_AUTHORITATIVE_INFORMATION
RC_NO_CONTENT
RC_RESET_CONTENT
RC_PARTIAL_CONTENT
RC_MULTIPLE_CHOICES
RC_MOVED_PERMANENTLY
RC_MOVED_TEMPORARILY
RC_SEE_OTHER
RC_NOT_MODIFIED
RC_USE_PROXY
RC_BAD_REQUEST
RC_UNAUTHORIZED
RC_PAYMENT_REQUIRED
RC_FORBIDDEN
RC_NOT_FOUND
RC_METHOD_NOT_ALLOWED
RC_NOT_ACCEPTABLE
RC_PROXY_AUTHENTICATION_REQUIRED
RC_REQUEST_TIMEOUT
RC_CONFLICT
RC_GONE
RC_LENGTH_REQUIRED
RC_PRECONDITION_FAILED
RC_REQUEST_ENTITY_TOO_LARGE
RC_REQUEST_URI_TOO_LARGE
RC_UNSUPPORTED_MEDIA_TYPE
RC_INTERNAL_SERVER_ERROR
RC_NOT_IMPLEMENTED
RC_BAD_GATEWAY
RC_SERVICE_UNAVAILABLE
RC_GATEWAY_TIMEOUT
RC_HTTP_VERSION_NOT_SUPPORTED
```

The HTTP::Status classification functions are:

is_success($rc)
: Check if response code indicated successfull request.

is_error($rc)
: Check if response code indicated that an error occured.

The module will also export the LWP::UserAgent object as `$ua` if you ask for it explicitly.

The user agent created by this module will identify itself as ‘‘LWP::Simple/0.00’’ and will initialize its proxy defaults from the environment (by calling `$ua`−>env_proxy).

SEE ALSO

the *LWP* manpage, the *LWP::UserAgent* manpage, the *HTTP::Status* manpage, the *request* manpage, the *mirror* manpage

NAME

LWP::Socket − TCP/IP socket interface

SYNOPSIS

```
$socket = new LWP::Socket;
$socket->connect('localhost', 7); # echo
$quote = 'I dunno, I dream in Perl sometimes...';
$socket->write("$quote\n");
$socket->read_until("\n", \$buffer);
$socket->read(\$buffer);
$socket = undef;  # close
```

DESCRIPTION

This class implements TCP/IP sockets. It groups socket generation, TCP address manipulation and buffered reading. Errors are handled by dying (throws exceptions).

This class should really not be required, something like this should be part of the standard Perl5 library.

Running this module standalone executes a self test which requires localhost to serve chargen and echo protocols.

METHODS

`$sock` = new *LWP::Socket()*

Constructs a new socket object.

`$sock`−>connect($host, `$port`)

Connect the socket to given host and port.

`$sock`−>*shutdown()*

Shuts down the connection.

`$sock`−>bind($host, `$port`)

Binds a name to the socket.

`$sock`−>*listen* ($queuesize)

Set up listen queue for socket.

`$sock`−>*accept* ($timeout)

Accepts a new connection. Returns a new LWP::Socket object if successful. Timeout not implemented yet.

`$sock`−>*getsockname()*

Returns a 2 element array ($host, `$port`)

`$sock`−>read_until($delim, `$data_ref`, `$size`, `$timeout`)

Reads data from the socket, up to a delimiter specified by a regular expression. If `$delim` is undefined all data is read. If `$size` is defined, data will be read internally in chunks of `$size` bytes. This does not mean that we will return the data when size bytes are read.

Note that `$delim` is discarded from the data returned.

`$sock->`read($bufref, [$size, `$timeout`])

Reads data of the socket. Not more than `$size` bytes. Might return less if the data is available. Dies on timeout.

`$sock->`*pushback* ($data)

Put data back into the socket. Data will returned next time you *read()*. Can be used if you find out that you have read too much.

`$sock->`write($data, [$timeout])

Write data to socket. The `$data` argument might be a scalar or code.

If data is a reference to a subroutine, then we will call this routine to obtain the data to be written. The routine will be called until it returns undef or empty data. Data might be returned from the callback as a scalar or as a reference to a scalar.

Write returns the number of bytes written to the socket.

_getaddress($h, `$p`)

Given a host and a port, it will return the address (sockaddr_in) suitable as the `name` argument for *connect()* or *bind()*. Might return several addresses in array context if the hostname is bound to several IP addresses.

SELF TEST

This self test is only executed when this file is run standalone. It tests its functions against some standard TCP services implemented by inetd. If you do not have them around the tests will fail.

NAME

LWP::TkIO − Tk I/O routines for the LWP library

SYNOPSIS

```
use Tk;
require LWP::TkIO;
require LWP::UserAgent;
```

DESCRIPTION

This module provide replacement functions for the LWP::IO functions. Require this module if you use Tk and want non exclusive IO behaviour from LWP. This does not allow LWP to run simultaneous requests though.

See also the *LWP::IO* manpage.

NAME

LWP::UserAgent − A WWW UserAgent class

SYNOPSIS

```
require LWP::UserAgent;
$ua = new LWP::UserAgent;

$request = new HTTP::Request('GET', 'file://localhost/etc/motd');

$response = $ua->request($request); # or
$response = $ua->request($request, '/tmp/sss'); # or
$response = $ua->request($request, \&callback, 4096);

sub callback { my($data, $response, $protocol) = @_; .... }
```

DESCRIPTION

The `LWP::UserAgent` is a class implementing a simple World-Wide Web user agent in Perl. It brings together the HTTP::Request, HTTP::Response and the LWP::Protocol classes that form the rest of the core of libwww-perl library. For simple uses this class can be used directly to dispatch WWW requests, alternatively it can be subclassed for application-specific behaviour.

In normal usage the application creates a UserAgent object, and then configures it with values for timeouts proxies, name, etc. The next step is to create an instance of `HTTP::Request` for the request that needs to be performed. This request is then passed to the UserAgent *request()* method, which dispatches it using the relevant protocol, and returns a `HTTP::Response` object.

The basic approach of the library is to use HTTP style communication for all protocol schemes, i.e. you will receive an `HTTP::Response` object also for gopher or ftp requests. In order to achieve even more similarities with HTTP style communications, gopher menus and file directories will be converted to HTML documents.

The *request()* method can process the content of the response in one of three ways: in core, into a file, or into repeated calls of a subroutine. You choose which one by the kind of value passed as the second argument to *request()*.

The in core variant simply returns the content in a scalar attribute called *content()* of the response object, and is suitable for small HTML replies that might need further parsing. This variant is used if the second argument is missing (or is undef).

The filename variant requires a scalar containing a filename as the second argument to *request()*, and is suitable for large WWW objects which need to be written directly to the file, without requiring large amounts of memory. In this case the response object returned from *request()* will have empty *content()*. If the request fails, then the *content()* might not be empty, and the file will be untouched.

The subroutine variant requires a reference to callback routine as the second argument to *request()* and it can also take an optional chuck size as third argument. This variant can be used to construct ‘‘pipe-lined’’ processing, where processing of received chuncks can begin before the complete data has arrived. The callback function is called with 3 arguments: the data received this time, a reference to the response object and a reference to the protocol object. The response object returned from *request()* will have empty *content()*. If the request fails, then the the callback routine will not have been called, and the response->*content()* might not be empty.

The request can be aborted by calling *die()* within the callback routine. The die message will be available as the ‘‘X−Died’’ special response header field.

The library also accepts that you put a subroutine reference as content in the request object. This subroutine should return the content (possibly in pieces) when called. It should return an empty string when there is no more content.

The user of this module can finetune timeouts and error handling by calling the *use_alarm()* and *use_eval()*

methods.

By default the library uses *alarm()* to implement timeouts, dying if the timeout occurs. If this is not the prefered behaviour or it interferes with other parts of the application one can disable the use alarms. When alarms are disabled timeouts can still occur for example when reading data, but other cases like name lookups etc will not be timed out by the library itself.

The library catches errors (such as internal errors and timeouts) and present them as HTTP error responses. Alternatively one can switch off this behaviour, and let the application handle dies.

SEE ALSO

See the *LWP* manpage for a complete overview of libwww-perl5. See the *request* manpage and the *mirror* manpage for examples of usage.

METHODS

`$ua` = new LWP::UserAgent;

Constructor for the UserAgent. Returns a reference to a LWP::UserAgent object.

`$ua`–>simple_request($request, [$arg [, `$size`]])

This method dispatches a single WWW request on behalf of a user, and returns the response received. The `$request` should be a reference to a `HTTP::Request` object with values defined for at least the *method()* and *url()* attributes.

If `$arg` is a scalar it is taken as a filename where the content of the response is stored.

If `$arg` is a reference to a subroutine, then this routine is called as chunks of the content is received. An optional `$size` argument is taken as a hint for an appropriate chunk size.

If `$arg` is omitted, then the content is stored in the response object itself.

`$ua`–>request($request, `$arg` [, `$size`])

Process a request, including redirects and security. This method may actually send several different simple reqeusts.

The arguments are the same as for `simple_request()`.

`$ua`–>redirect_ok

This method is called by *request()* before it tries to do any redirects. It should return a true value if the redirect is allowed to be performed. Subclasses might want to override this.

The default implementation will return FALSE for POST request and TRUE for all others.

`$ua`–>credentials($netloc, `$realm`, `$uname`, `$pass`)

Set the user name and password to be used for a realm. It is often more useful to specialize the *get_basic_credentials()* method instead.

`$ua`–>get_basic_credentials($realm, `$uri`)

This is called by *request()* to retrieve credentials for a Realm protected by Basic Authentication or Digest Authentication.

Should return username and password in a list. Return undef to abort the authentication resolution atempts.

This implementation simply checks a set of pre-stored member variables. Subclasses can override this method to e.g. ask the user for a username/password. An example of this can be found in `lwp-request` program distributed with this library.

$ua->*agent* ([$product_id])

Get/set the product token that is used to identify the user agent on the network. The agent value is sent as the ''User-Agent'' header in the requests. The default agent name is ''libwww-perl/#.##'', where ''#.##'' is substitued with the version numer of this library.

The user agent string should be one or more simple product identifiers with an optional version number separated by the ''/'' character. Examples are:

```
$ua->agent('Checkbot/0.4 ' . $ua->agent);
$ua->agent('Mozilla/5.0');
```

$ua->*from* ([$email_address])

Get/set the Internet e-mail address for the human user who controls the requesting user agent. The address should be machine-usable, as defined in RFC 822. The from value is send as the ''From'' header in the requests. There is no default. Example:

```
$ua->from('aas@sn.no');
```

$ua->*timeout* ([$secs])

Get/set the timeout value in seconds. The default *timeout()* value is 180 seconds, i.e. 3 minutes.

$ua->*use_alarm* ([$boolean])

Get/set a value indicating wether to use *alarm()* when implementing timeouts. The default is TRUE, i.e. to use alarm. Disable this on systems that does not implement alarm, or if this interfers with other uses of alarm in your application.

$ua->*use_eval* ([$boolean])

Get/set a value indicating wether to handle internal errors internally by trapping with eval. The default is TRUE, i.e. the $ua->*request()* will never die.

$ua->*parse_head* ([$boolean])

Get/set a value indicating wether we should initialize response headers from the <head> section of HTML documents. The default is TRUE. Do not turn this off, unless you know what you are doing.

$ua->*max_size* ([$bytes])

Get/set the size limit for response content. The default is undef, which means that there is not limit. If the returned response content is only partial, because the size limit was exceeded, then a ''X–Content-Range'' header will be added to the response.

$ua->clone;

Returns a copy of the LWP::UserAgent object

$ua->*is_protocol_supported* ($scheme)

You can use this method to query if the library currently support the specified `scheme`. The `scheme` might be a string (like 'http' or 'ftp') or it might be an URI::URL object reference.

$ua->mirror($url, `$file`)

Get and store a document identified by a URL, using If-Modified-Since, and checking of the Content-Length. Returns a reference to the response object.

$ua->*proxy* (...)

Set/retrieve proxy URL for a scheme:

```
$ua->proxy(['http', 'ftp'], 'http://proxy.sn.no:8001/');
$ua->proxy('gopher', 'http://proxy.sn.no:8001/');
```

The first form specifies that the URL is to be used for proxying of access methods listed in the list in the first method argument, i.e. 'http' and 'ftp'.

The second form shows a shorthand form for specifying proxy URL for a single access scheme.

$ua->*env_proxy()*

Load proxy settings from *_proxy environment variables. You might specify proxies like this (sh-syntax):

```
gopher_proxy=http://proxy.my.place/
wais_proxy=http://proxy.my.place/
no_proxy="my.place"
export gopher_proxy wais_proxy no_proxy
```

Csh or tcsh users should use the `setenv` command to define these envirionment variables.

$ua->no_proxy($domain,...)

Do not proxy requests to the given domains. Calling no_proxy without any domains clears the list of domains. Eg:

```
$ua->no_proxy('localhost', 'no', ...);
```

NAME

encode_base64 − Encode string using base64 encoding

decode_base64 − Decode base64 string

SYNOPSIS

```
 use MIME::Base64;

 $encoded = encode_base64('Aladdin:open sesame');
 $decoded = decode_base64($encoded);
```

DESCRIPTION

This module provides functions to encode and decode strings into the Base64 encoding specified in RFC 1521 − *MIME (Multipurpose Internet Mail Extensions)*. The Base64 encoding is designed to represent arbitrary sequences of octets in a form that need not be humanly readable. A 65-character subset ([A−Za-z0-9+/=]) of US−ASCII is used, enabling 6 bits to be represented per printable character.

RFC 1521 says that the encoded bytes must be represented in lines of no more than 76 characters each. The second argument to *encode_base64()* is the line ending sequence to use. It defaults to `"\n"`. Use an empty string as second argument if you do not want the encoded string broken into lines.

If you prefer not to import these routines into your namespace you can call them as:

```
  use MIME::Base64 ();
  $encoded = MIME::Base64::encode('Aladdin:open sesame');
  $decoded = MIME::Base64::decode($encoded);
```

COPYRIGHT

AUTHOR

Gisle Aas <aas@sn.no>

Based on LWP::Base64 written by Martijn Koster <m.koster@nexor.co.uk> and Joerg Reichelt <j.reichelt@nexor.co.uk> and code posted to comp.lang.perl <3pd2lp$6gf@wsinti07.win.tue.nl> by Hans Mulder <hansm@wsinti07.win.tue.nl>

NAME

encode_qp – Encode string using quoted-printable encoding

decode_qp – Decode quoted-printable string

SYNOPSIS

```
use MIME::QuotedPrint;

$encoded = encode_qp($decoded);
$decoded = decode_qp($encoded);
```

DESCRIPTION

This module provides functions to encode and decode strings into the Quoted-Printable encoding specified in RFC 1521 – *MIME (Multipurpose Internet Mail Extensions)*. The Quoted-Printable encoding is intended to represent data that largely consists of bytes that correspond to printable characters in the ASCII character set. Non-printable characters (as defined by enghlish americans) are represented by a triplet consisting of the character "=" followed by two hexadecimal digits.

Note that the *encode_qp()* routine does not change newlines `"\n"` to the CRLF sequence even though this might be considered the right thing to do (RFC 1521 (Q–P Rule #4)).

If you prefer not to import these routines into your namespace you can call them as:

```
use MIME::QuotedPrint ();
$encoded = MIME::QuotedPrint::encode($decoded);
$decoded = MIME::QuotedPrint::decode($encoded);
```

COPYRIGHT

AUTHOR

Gisle Aas <aas@sn.no>

NAME

uri_escape − Escape unsafe characters

uri_unescape − Unescape escaped characters

SYNOPSIS

```
 use URI::Escape;
 $safe = uri_escape("10% is enough\n");
 $verysafe = uri_escape("foo", "\0-\377");
 $str  = uri_unescape($safe);
```

DESCRIPTION

This module provide functions to escape and unescape URI strings. Some characters are regarded as "unsafe" and must be escaped in accordance with RFC 1738. Escaped characters are represented by a triplet consisting of the character "%" followed by two hexadecimal digits.

The *uri_escape()* function takes an optional second argument that overrides the set of characters that are to be escaped. The set is specified as a string that can be used in a regular expression character class (between [ ]). E.g.:

```
  \x00-\x1f\x7f-\xff          # all control and hi-bit characters
  a-z                         # all lower case characters
  ^A-Za-z                     # everything not a letter
```

The default set of characters to be escaped is:

```
  \x00-\x20"#%;<>?{}|\\\\^~`\[\]\x7F-\xFF
```

The module can also export the `%escapes` hash which contains the mapping from all characters to the corresponding escape code.

SEE ALSO

the *URI::URL* manpage

NAME

URI::URL − Uniform Resource Locators (absolute and relative)

SYNOPSIS

```
use URI::URL;

# Constructors
$url1 = new URI::URL 'http://www.perl.com/%7Euser/gisle.gif';
$url2 = new URI::URL 'gisle.gif', 'http://www.com/%7Euser';
$url3 = url 'http://www.sn.no/'; # handy constructor
$url4 = $url2->abs;       # get absolute url using base
$url5 = $url2->abs('http:/other/path');
$url6 = newlocal URI::URL 'test';

# Stringify URL
$str1 = $url->as_string;  # complete escaped URL string
$str2 = $url->full_path;  # escaped path+params+query
$str3 = "$url";           # use operator overloading

# Retrieving Generic-RL components:
$scheme   = $url->scheme;
$netloc   = $url->netloc; # see user,password,host,port below
$path     = $url->path;
$params   = $url->params;
$query    = $url->query;
$frag     = $url->frag;

# Accessing elements in their escaped form
$path     = $url->epath;
$params   = $url->eparams;
$query    = $url->equery;

# Retrieving Network location (netloc) components:
$user     = $url->user;
$password = $url->password;
$host     = $url->host;
$port     = $url->port;   # returns default if not defined

# Retrieve escaped path components as an array
@path     = $url->path_components;

# HTTP query-string access methods
@keywords = $url->keywords;
@form     = $url->query_form;

# All methods above can set the field values, e.g:
$url->scheme('http');
$url->host('www.w3.org');
$url->port($url->default_port);
$url->base($url5);                      # use string or object
$url->keywords(qw(dog bones));
```

```
# File methods
$url = new URI::URL "file:/foo/bar";
open(F, $url->local_path) or die;

# Compare URLs
if ($url->eq("http://www.sn.no")) or die;
```

DESCRIPTION

This module implements the URI::URL class representing Uniform Resource Locators (URL). URLs provide a compact string representation for resources available via the Internet. Both absolute (RFC 1738) and relative (RFC 1808) URLs are supported.

URI::URL objects are created by calling *new()*, which takes as argument a string representation of the URL or an existing URL object reference to be cloned. Specific individual elements can then be accessed via the *scheme()*, *user()*, *password()*, *host()*, *port()*, *path()*, *params()*, *query()* and *frag()* methods. In addition escaped versions of the path, params and query can be accessed with the *epath()*, *eparams()* and *equery()* methods. Note that some URL schemes will support all these methods.

The object constructor *new()* must be able to determine the scheme for the URL. If a scheme is not specified in the URL itself, it will use the scheme specified by the base URL. If no base URL scheme is defined then *new()* will croak if URI::URL::*strict*(1) has been invoked, otherwise *http* is silently assumed. Once the scheme has been determined *new()* then uses the *implementor()* function to determine which class implements that scheme. If no implementor class is defined for the scheme then *new()* will croak if URI::URL::*strict*(1) has been invoked, otherwise the internal generic URL class is assumed.

Internally defined schemes are implemented by the URI::URL::*scheme_name* module. The *URI::URL::implementor()* function can be used to explicitly set the class used to implement a scheme if you want to override this.

HOW AND WHEN TO ESCAPE

This is an edited extract from a URI specification:

> The printability requirement has been met by specifying a safe set of characters, and a general escaping scheme for encoding "unsafe" characters. This "safe" set is suitable, for example, for use in electronic mail. This is the canonical form of a URI.
>
> There is a conflict between the need to be able to represent many characters including spaces within a URI directly, and the need to be able to use a URI in environments which have limited character sets or in which certain characters are prone to corruption. This conflict has been resolved by use of an hexadecimal escaping method which may be applied to any characters forbidden in a given context. When URLs are moved between contexts, the set of characters escaped may be enlarged or reduced unambiguously. The canonical form for URIs has all white spaces encoded.

Notes:

> A URL string *must*, by definition, consist of escaped components. Complete URLs are always escaped.
>
> The components of a URL string must be *individually* escaped. Each component of a URL may have a separate requirements regarding what must be escaped, and those requirements are also dependent on the URL scheme.
>
> Never escape an already escaped component string.

This implementation expects an escaped URL string to be passed to *new()* and will return a fully escaped URL string from *as_string()* and *full_path()*.

Individual components can be manipulated in unescaped or escaped form. The following methods return/accept unescaped strings:

```
scheme                  path
user                    params
password                query
host                    frag
port
```

The following methods return/accept partial *escaped* strings:

```
netloc                  eparams
epath                   equery
```

Partial escaped means that only reserved characters (i.e. ':', '@', '/', ';', '?', '=', '&' in addition to '%', '.' and '#') needs to be escaped when they are to be treated as normal characters. *Fully escaped* means that all unsafe characters are escaped. Unsafe characters are all all control characters (%00-%1F and `%7F`), all 8-bit characters (%80-%FF) as well as '{', '}', '|', '\', '^', '[', ']' '`', '"', '<' and '>'. Note that the character '~' is **not** considered unsafe by this library as it is common practice to use it to reference personal home pages, but it is still unsafe according to RFC 1738.

ADDING NEW URL SCHEMES

New URL schemes or alternative implementations for existing schemes can be added to your own code. To create a new scheme class use code like:

```
package MYURL::foo;
@ISA = (URI::URL::implementor());   # inherit from generic scheme
```

The '*URI::URL::implementor()*' function call with no parameters returns the name of the class which implements the generic URL scheme behaviour (typically `URI::URL::_generic`). All hierarchical schemes should be derived from this class.

Your class can then define overriding methods (e.g., *new()*, *_parse()* as required).

To register your new class as the implementor for a specific scheme use code like:

```
URI::URL::implementor('x-foo', 'MYURL::foo');
```

Any new URL created for scheme 'x-foo' will be implemented by your `MYURL::foo` class. Existing URLs will not be affected.

FUNCTIONS

new URI::URL $url_string [, $base_url]
: This is the object constructor. It will create a new URI::URL object, initialized from the URL string. To trap bad or unknown URL schemes use:

```
$obj = eval { new URI::URL "snews:comp.lang.perl.misc" };
```

or set URI::URL::*strict*(0) if you do not care about bad or unknown schemes.

newlocal URI::URL $path;
: Returns an URL object that denotes a path within the local filesystem. Paths not starting with '/' are interpreted relative to the current working directory. This constructor always return an absolute 'file' URL.

url($url_string, [, $base_url])
: Alternative constructor function. The *url()* function is exported by the URI::URL module and is easier both to type and read than calling URI::URL->new directly. Useful for constructs like this:

```
$h = url($str)->host;
```

This function is just a wrapper for URI::URL->new.

URI::URL::strict($bool)
: If strict is true then we croak on errors. The function returns the previous value.

URI::URL::implementor([$scheme, [$class]])
: Use this function to get or set implementor class for a scheme. Returns '' if specified scheme is not supported. Returns generic URL class if no scheme specified.

METHODS

This section describes the methods available for an URI::URL object. Note that some URL schemes will disallow some of these methods and will croak if they are used. Some URL schemes add additional methods that are described in the sections to follow.

Attribute access methods marked with (*) can take an optional argument to set the value of the attribute, and they always return the old value.

$url->abs([$base, [$allow_scheme_in_relative_urls]])
: The *abs()* method attempts to return a new absolute URI::URL object for a given URL. In order to convert a relative URL into an absolute one, a *base* URL is required. You can associate a default base with a URL either by passing a *base* to the *new()* constructor when a URI::URL is created or using the *base()* method on the object later. Alternatively you can specify a one-off base as a parameter to the *abs()* method.

  Some older parsers used to allow the scheme name to be present in the relative URL if it was the same as the base URL scheme. RFC1808 says that this should be avoided, but you can enable this old behaviour by passing a TRUE value as the second argument to the *abs()* method. The difference is demonstrated by the following examples:

```
url("http:foo")->abs("http://host/a/b")      ==>  "http:foo"
url("http:foo")->abs("http://host/a/b", 1)  ==>  "http:/host/a/foo"
```

  The *rel()* method will do the opposite transformation.

$url->as_string
: Returns a string representing the URL in its canonical form. All unsafe characters will be escaped. This method is overloaded as the perl "stringify" operator, which means that URLs can be used as strings in many contexts.

$url->base (*)
: Get/set the base URL associated with the current URI::URL object. The base URL matters when you call the *abs()* method.

$url->clone
: Returns a copy of the current URI::URL object.

$url->crack
: Return a 9 element array with the following content:

```
0: $url->scheme *)
1: $url->user
2: $url->password
3: $url->host
4: $url->port
5: $url->epath
6: $url->eparams
7: $url->equery
8: $url->frag
```

  All elements except *scheme* will be undefined if the corresponding URL part is not available.

  Note: The scheme (first element) returned by crack will aways be defined. This is different from what the `$url->`scheme returns, since it will return *undef* for relative URLs.

$url->default_port
Returns the default port number for the URL scheme that the URI::URL belongs too.

$url->eparams (*)
Get/set the URL parameters in escaped form.

$url->epath (*)
Get/set the URL path in escaped form.

$url->eq($other_url)
Compare two URLs to decide if they match or not. The rules for how comparison is made varies for different parts of the URLs; scheme and netloc comparison is case-insensitive, and escaped chars match their `%XX` encoding unless they are "reserved" or "unsafe".

$url->equery (*)
Get/set the URL query string in escaped form.

$url->full_path
Returns the string "/path;params?query". This is the string that is passed to a remote server in order to access the document.

$url->frag (*)
Get/set the fragment (unescaped)

$url->host (*)
Get/set the host (unescaped)

$url->netloc (*)
Get/set the network location in escaped form. Setting the network location will affect 'user', 'password', 'host' and 'port'.

$url->params (*)
Get/set the URL parameters (unescaped)

$url->password (*)
Get/set the password (unescaped)

$url->path (*)
Get/set the path (unescaped). This method will croak if any of the path components in the return value contain the "/" character. You should use the *epath()* method to be safe.

$url->path_components (*)
Get/set the path using a list of unescaped path components. The return value will loose the distinction beween '.' and '%2E'. When setting a value, a '.' is converted to be a literal '.' and is therefore encoded as '%2E'.

$url->port (*)
Get/set the network port (unescaped)

$url->rel([$base])
Return a relative URL if possible. This is the opposite of what the *abs()* method does. For instance:

```
url("http://www.math.uio.no/doc/mail/top.html",
    "http://www.math.uio.no/doc/linux/")->rel
```

will return a relative URL with path set to "../mail/top.html" and with the same base as the original URL.

If the original URL already is relative or the scheme or netloc does not match the base, then a copy of the original URL is returned.

$url->print_on(*FILEHANDLE);
Prints a verbose presentation of the contents of the URL object to the specified file handle (default STDOUT). Mainly useful for debugging.

$url->scheme (*)
Get/set the scheme for the URL.

$url->query (*)
Get/set the query string (unescaped). This method will croak if the string returned contains both '+' and '%2B' or '=' together with '%3D' or '%26'. You should use the *equery()* method to be safe.

$url->user (*)
Get/set the URL user name (unescaped)

HTTP METHODS

For *http* URLs you may also access the query string using the *keywords()* and the *query_form()* methods. Both will croak if the query is not of the correct format. The encodings look like this:

```
word1+word2+word3..        # keywords
key1=val1&key2=val2...     # query_form
```

Note: These functions does not return the old value when they are used to set a value of the query string.

$url->keywords (*)
The *keywords()* method returns a list of unescaped strings. The method can also be used to set the query string by passing in the keywords as individual arguments to the method.

$url->query_form (*)
The *query_form()* method return a list of unescaped key/value pairs. If you assign the return value to a hash you might loose some values if the key is repeated (which it is allowed to do).

This method can also be used to set the query sting of the URL like this:

```
$url->query_form(foo => 'bar', foo => 'baz', equal => '=');
```

If the value part of a key/value pair is a reference to an array, then it will be converted to separate key/value pairs for each value. This means that these two calls are equal:

```
$url->query_form(foo => 'bar', foo => 'baz');
$url->query_form(foo => ['bar', 'baz']);
```

FILE METHODS

The *file* URLs implement the *local_path()* method that returns a path suitable for access to files within the current filesystem. These methods can **not** be used to set the path of the URL.

$url->local_path
This method is really just an alias for one of the methods below depending on what system you run on.

$url->unix_path
Returns a path suitable for use on a Unix system. This method will croak if any of the path segments contains a "/" or a NULL character.

$url->dos_path
Returns a path suitable for use on a MS−DOS or MS−Windows system.

$url->mac_path
Returns a path suitable for use on a Macintosh system.

$url->vms_path
Returns a path suitable for use on a VMS system. VMS is a trademark of Digital.

GOPHER METHODS

The methods access the parts that are specific for the gopher URLs. These methods access different parts of the `$url`−>path.

$url->gtype (*)

$url->selector (*)

$url->search (*)

$url->string (*)

NEWS METHODS

$url->group (*)

$url->article (*)

WAIS METHODS

The methods access the parts that are specific for the wais URLs. These methods access different parts of the `$url->`path.

$url->database (*)

$url->wtype (*)

$url->wpath (*)

MAILTO METHODS

$url->address (*)
The mail address can also be accessed with the *netloc()* method.

WHAT A URL IS NOT

URL objects do not, and should not, know how to 'get' or 'put' the resources they specify locations for, anymore than a postal address 'knows' anything about the postal system. The actual access/transfer should be achieved by some form of transport agent class (see the *LWP::UserAgent* manpage). The agent class can use the URL class, but should not be a subclass of it.

COMPATIBILITY

This is a listing incompatibilities with URI::URL version 3.x:

unsafe(), escape() and unescape()
These methods not supported any more.

full_path() and as_string()
These methods does no longer take a second argument which specify the set of characters to consider as unsafe.

'+' in the query-string
The '+' character in the query part of the URL was earlier considered to be an encoding of a space. This was just bad influence from Mosaic. Space is now encoded as '%20'.

path() and query()
This methods will croak if they loose information. Use *epath()* or *equery()* instead. The *path()* method will for instance loose information if any path segment contain an (encoded) '/' character.

The *path()* now consider a leading '/' to be part of the path. If the path is empty it will default to '/'. You can get the old behaviour by setting `$URI::URL::COMPAT_VER_3` to TRUE before accessing the *path()* method.

netloc()
The string passed to netloc is now assumed to be escaped. The string returned will also be (partially) escaped.

sub-classing
The path, params and query is now stored internally in unescaped form. This might affect sub-classes of the URL scheme classes.

AUTHORS / ACKNOWLEDGMENTS

This module is (distantly) based on the `wwwurl.pl` code in the libwww-perl distribution developed by Roy Fielding <fielding@ics.uci.edu>, as part of the Arcadia project at the University of California, Irvine,

with contributions from Brooks Cutter.

Gisle Aas <aas@sn.no>, Tim Bunce <Tim.Bunce@ig.co.uk>, Roy Fielding <fielding@ics.uci.edu> and Martijn Koster <m.koster@webcrawler.com> (in English alphabetical order) have collaborated on the complete rewrite for Perl 5, with input from other people on the libwww-perl mailing list.

If you have any suggestions, bug reports, fixes, or enhancements, send them to the libwww-perl mailing list at <libwww-perl@ics.uci.edu>.

COPYRIGHT

NAME

WWW::RobotsRules – Parse robots.txt files

SYNOPSIS

```
require WWW::RobotRules;
my $robotsrules = new WWW::RobotRules 'MOMspider/1.0';

use LWP::Simple qw(get);

$url = "http://some.place/robots.txt";
my $robots_txt = get $url;
$robotsrules->parse($url, $robots_txt);

$url = "http://some.other.place/robots.txt";
my $robots_txt = get $url;
$robotsrules->parse($url, $robots_txt);

# Now we are able to check if a URL is valid for those servers that
# we have obtained and parsed "robots.txt" files for.
if($robotsrules->allowed($url)) {
    $c = get $url;
    ...
}
```

DESCRIPTION

This module parses a *robots.txt* file as specified in "A Standard for Robot Exclusion", described in <URL:http://info.webcrawler.com/mak/projects/robots/norobots.html> Webmasters can use the *robots.txt* file to disallow conforming robots access to parts of their WWW server.

The parsed file is kept in the WWW::RobotRules object, and this object provide methods to check if access to a given URL is prohibited. The same WWW::RobotRules object can parse multiple *robots.txt* files.

METHODS

`$rules` = new WWW::RobotRules 'MOMspider/1.0'

This is the constructor for WWW::RobotRules objects. The first argument given to *new()* is the name of the robot.

`$rules->`parse($url, `$content`, `$fresh_until`)

The *parse()* method takes as arguments the URL that was used to retrieve the */robots.txt* file, and the contents of the file.

`$rules->`*allowed* ($url)

Returns TRUE if this robot is allowed to retrieve this URL.

`$rules->`*agent* ([$name])

Get/set the agent name. NOTE: Changing the agent name will clear the robots.txt rules and expire times out of the cache.

ROBOTS.TXT

The format and semantics of the "/robots.txt" file are as follows (this is an edited abstract of <URL:http://info.webcrawler.com/mak/projects/robots/norobots.html>):

The file consists of one or more records separated by one or more blank lines. Each record contains lines of the form

```
<field-name>: <value>
```

The field name is case insensitive. Text after the '#' character on a line is ignored during parsing. This is used for comments. The following <field-names> can be used:

User-Agent
: The value of this field is the name of the robot the record is describing access policy for. If more than one *User-Agent* field is present the record describes an identical access policy for more than one robot. At least one field needs to be present per record. If the value is '*', the record describes the default access policy for any robot that has not not matched any of the other records.

Disallow
: The value of this field specifies a partial URL that is not to be visited. This can be a full path, or a partial path; any URL that starts with this value will not be retrieved

Examples

The following example "/robots.txt" file specifies that no robots should visit any URL starting with "/cyberworld/map/" or "/tmp/":

```
# robots.txt for http://www.site.com/

User-agent: *
Disallow: /cyberworld/map/ # This is an infinite virtual URL space
Disallow: /tmp/ # these will soon disappear
```

This example "/robots.txt" file specifies that no robots should visit any URL starting with "/cyberworld/map/", except the robot called "cybermapper":

```
# robots.txt for http://www.site.com/

User-agent: *
Disallow: /cyberworld/map/ # This is an infinite virtual URL space

# Cybermapper knows where to go.
User-agent: cybermapper
Disallow:
```

This example indicates that no robots should visit this site further:

```
# go away
User-agent: *
Disallow: /
```

SEE ALSO

the *LWP::RobotUA* manpage, the *WWW::RobotRules::AnyDBM_File* manpage

NAME

WWW::RobotRules::AnyDBM_File – Persistent RobotRules

SYNOPSIS

```
require WWW::RobotRules::AnyDBM_File;
require LWP::RobotUA;

# Create a robot useragent that uses a diskcaching RobotRules
my $rules = new WWW::RobotRules::AnyDBM_File 'my-robot/1.0', 'cachefile';
my $ua = new WWW::RobotUA 'my-robot/1.0', 'me@foo.com', $rules;

# Then just use $ua as usual
$res = $ua->request($req);
```

DESCRIPTION

This is a subclass of *WWW::RobotRules* that uses the AnyDBM_File package to implement persistent diskcaching of *robots.txt* and host visit information.

The constructor (the *new()* method) takes an extra argument specifying the name of the DBM file to use. If the DBM file already exists, then you can specify undef as agent name as the name can be obtained from the DBM database.

SE ALSO

the *WWW::RobotRules* manpage, the *LWP::RobotUA* manpage

AUTHORS

Hakan Ardo <hakan@munin.ub2.lu.se>, Gisle Aas <aas@sn.no>

NAME

Net::Cmd – Network Command class (as used by FTP, SMTP etc)

SYNOPSIS

```
use Net::Cmd;

@ISA = qw(Net::Cmd);
```

DESCRIPTION

`Net::Cmd` is a collection of methods that can be inherited by a sub class of `IO::Handle`. These methods implement the functionality required for a command based protocol, for example FTP and SMTP.

USER METHODS

These methods provide a user interface to the `Net::Cmd` object.

debug (VALUE)
: Set the level of debug information for this object. If `VALUE` is not given then the current state is returned. Otherwise the state is changed to `VALUE` and the previous state returned.

  Set the level of debug information for this object. If no argument is given then the current state is returned. Otherwise the state is changed to `$value`and the previous state returned. Different packages may implement different levels of debug but, a non-zero value result in copies of all commands and responses also being sent to STDERR.

  If `VALUE` is `undef` then the debug level will be set to the default debug level for the class.

  This method can also be called as a *static* method to set/get the default debug level for a given class.

message ()
: Returns the text message returned from the last command

code ()
: Returns the 3-digit code from the last command. If a command is pending then the value 0 is returned

ok ()
: Returns non-zero if the last code value was greater than zero and less than 400. This holds true for most command servers. Servers where this does not hold may override this method.

status ()
: Returns the most significant digit of the current status code. If a command is pending then `CMD_PENDING` is returned.

datasend (DATA)
: Send data to the remote server, converting LF to CRLF. Any line starting with a '.' will be prefixed with another '.'. `DATA` may be an array or a reference to an array.

dataend ()
: End the sending of data to the remote server. This is done by ensuring that the data already sent ends with CRLF then sending '.CRLF' to end the transmission. Once this data has been sent `dataend` calls `response` and returns true if `response` returns CMD_OK.

CLASS METHODS

These methods are not intended to be called by the user, but used or over-ridden by a sub-class of `Net::Cmd`

debug_print (DIR, TEXT)
: Print debugging information. `DIR` denotes the direction *true* being data being sent to the server. Calls `debug_text` before printing to STDERR.

debug_text (TEXT)
: This method is called to print debugging information. TEXT is the text being sent. The method should return the text to be printed

This is primarily meant for the use of modules such as FTP where passwords are sent, but we do not want to display them in the debugging information.

command (CMD [, ARGS, ...])
: Send a command to the command server. All arguments a first joined with a space character and CRLF is appended, this string is then sent to the command server.

  Returns undef upon failure

unsupported ()
: Sets the status code to 580 and the response text to 'Unsupported command'. Returns zero.

response ()
: Obtain a response from the server. Upon success the most significant digit of the status code is returned. Upon failure, timeout etc., *undef* is returned.

parse_response (TEXT)
: This method is called by `response` as a method with one argument. It should return an array of 2 values, the 3-digit status code and a flag which is true when this is part of a multi-line response and this line is not the list.

getline ()
: Retrieve one line, delimited by CRLF, from the remote server. Returns *undef* upon failure.

  NOTE: If you do use this method for any reason, please remember to add some `debug_print` calls into your method.

ungetline (TEXT)
: Unget a line of text from the server.

read_until_dot ()
: Read data from the remote server until a line consisting of a single '.'. Any lines starting with '..' will have one of the '.'s removed.

  Returns a reference to a list containing the lines, or *undef* upon failure.

EXPORTS

`Net::Cmd` exports six subroutines, five of these, `CMD_INFO`, `CMD_OK`, `CMD_MORE`, `CMD_REJECT` and `CMD_ERROR` ,correspond to possible results of `response` and `status`. The sixth is `CMD_PENDING`.

AUTHOR

Graham Barr <gbarr@ti.com>

COPYRIGHT

NAME

Net::Domain − Attempt to evaluate the current host's internet name and domain

SYNOPSIS

```
use Net::Domain qw(hostname hostfqdn hostdomain);
```

DESCRIPTION

Using various methods **attempt** to find the Fully Qualified Domain Name (FQDN) of the current host. From this determine the host-name and the host-domain.

Each of the functions will return *undef* if the FQDN cannot be determined.

hostfqdn ()
: Identify and return the FQDN of the current host.

hostname ()
: Returns the smallest part of the FQDN which can be used to identify the host.

hostdomain ()
: Returns the remainder of the FQDN after the *hostname* has been removed.

AUTHOR

Graham Barr <gbarr@ti.com>. Adapted from Sys::Hostname by David Sundstrom <sunds@asictest.sc.ti.com>

COPYRIGHT

NAME

Net::DummyInetd – A dummy Inetd server

SYNOPSIS

```
use Net::DummyInetd;
use Net::SMTP;

$inetd = new Net::DummyInetd qw(/usr/lib/sendmail -ba -bs);

$smtp  = Net::SMTP->new('localhost', Port => $inetd->port);
```

DESCRIPTION

`Net::DummyInetd` is just what it's name says, it is a dummy inetd server. Creation of a `Net::DummyInetd` will cause a child process to be spawned off which will listen to a socket. When a connection arrives on this socket the specified command is fork'd and exec'd with STDIN and STDOUT file descriptors duplicated to the new socket.

This package was added as an example of how to use `Net::SMTP` to connect to a `sendmail` process, which is not the default, via SIDIN and STDOUT. A `Net::Inetd` package will be available in the next release of `libnet`

CONSTRUCTOR

new (CMD)

Creates a new object and spawns a child process which listens to a socket. `CMD` is a list, which will be passed to `exec` when a new process needs to be created.

METHODS

port

Returns the port number on which the *DummyInetd* object is listening

AUTHOR

Graham Barr <gbarr@ti.com>

COPYRIGHT

NAME

Net::FTP – FTP Client class

SYNOPSIS

```
use Net::FTP;

$ftp = Net::FTP->new("some.host.name");
$ftp->login("anonymous","me@here.there");
$ftp->cwd("/pub");
$ftp->get("that.file");
$ftp->quit;
```

DESCRIPTION

`Net::FTP` is a class implementing a simple FTP client in Perl as described in RFC959. It provides wrappers for a subset of the RFC959 commands.

OVERVIEW

FTP stands for File Transfer Protocol. It is a way of transferring files between networked machines. The protocol defines a client (whose commands are provided by this module) and a server (not implemented in this module). Communication is always initiated by the client, and the server responds with a message and a status code (and sometimes with data).

The FTP protocol allows files to be sent to or fetched from the server. Each transfer involves a **local file** (on the client) and a **remote file** (on the server). In this module, the same file name will be used for both local and remote if only one is specified. This means that transferring remote file `/path/to/file` will try to put that file in `/path/to/file` locally, unless you specify a local file name.

The protocol also defines several standard **translations** which the file can undergo during transfer. These are ASCII, EBCDIC, binary, and byte. ASCII is the default type, and indicates that the sender of files will translate the ends of lines to a standard representation which the receiver will then translate back into their local representation. EBCDIC indicates the file being transferred is in EBCDIC format. Binary (also known as image) format sends the data as a contiguous bit stream. Byte format transfers the data as bytes, the values of which remain the same regardless of differences in byte size between the two machines (in theory – in practice you should only use this if you really know what you're doing).

CONSTRUCTOR

new (HOST [,OPTIONS])

This is the constructor for a new Net::FTP object. `HOST` is the name of the remote host to which a FTP connection is required.

`OPTIONS` are passed in a hash like fashion, using key and value pairs. Possible options are:

Firewall – The name of a machine which acts as a FTP firewall. This can be overridden by an environment variable `FTP_FIREWALL`. If specified, and the given host cannot be directly connected to, then the connection is made to the firewall machine and the string `@hostname` is appended to the login identifier. This kind of setup is also refered to as a ftp proxy.

Port – The port number to connect to on the remote machine for the FTP connection

Timeout – Set a timeout value (defaults to 120)

Debug – debug level (see the debug method in the *Net::Cmd* manpage)

Passive – If set to *true* then all data transfers will be done using passive mode. This is required for some *dumb* servers, and some firewall configurations. This can also be set by the environment variable `FTP_PASSIVE`.

METHODS

Unless otherwise stated all methods return either a *true* or *false* value, with *true* meaning that the operation was a success. When a method states that it returns a value, failure will be returned as *undef* or an empty list.

login ([LOGIN [,PASSWORD [, ACCOUNT]]])
: Log into the remote FTP server with the given login information. If no arguments are given then the `Net::FTP` uses the `Net::Netrc` package to lookup the login information for the connected host. If no information is found then a login of *anonymous* is used. If no password is given and the login is *anonymous* then the users Email address will be used for a password.

  If the connection is via a firewall then the `authorize` method will be called with no arguments.

authorize ([AUTH [, RESP]])
: This is a protocol used by some firewall ftp proxies. It is used to authorise the user to send data out. If both arguments are not specified then `authorize` uses `Net::Netrc` to do a lookup.

type (TYPE [, ARGS])
: This method will send the TYPE command to the remote FTP server to change the type of data transfer. The return value is the previous value.

ascii ([ARGS]) binary([ARGS]) ebcdic([ARGS]) byte([ARGS])
: Synonyms for `type` with the first arguments set correctly

  NOTE ebcdic and byte are not fully supported.

rename (OLDNAME, NEWNAME)
: Rename a file on the remote FTP server from `OLDNAME` to `NEWNAME`. This is done by sending the RNFR and RNTO commands.

delete (FILENAME)
: Send a request to the server to delete `FILENAME`.

cwd ([DIR])
: Attempt to change directory to the directory given in `$dir`. If `$dir` is `".."`, the FTP `CDUP` command is used to attempt to move up one directory. If no directory is given then an attempt is made to change the directory to the root directory.

cdup ()
: Change directory to the parent of the current directory.

pwd ()
: Returns the full pathname of the current directory.

rmdir (DIR)
: Remove the directory with the name `DIR`.

mkdir (DIR [, RECURSE])
: Create a new directory with the name `DIR`. If `RECURSE` is *true* then `mkdir` will attempt to create all the directories in the given path.

  Returns the full pathname to the new directory.

ls ([DIR])
: Get a directory listing of `DIR`, or the current directory.

  Returns a reference to a list of lines returned from the server.

dir ([DIR])
: Get a directory listing of `DIR`, or the current directory in long format.

  Returns a reference to a list of lines returned from the server.

get (REMOTE_FILE [, LOCAL_FILE])
: Get `REMOTE_FILE` from the server and store locally. `LOCAL_FILE` may be a filename or a filehandle. If not specified the the file will be stored in the current directory with the same leafname as the remote file.

  Returns `LOCAL_FILE`, or the generated local file name if `LOCAL_FILE` is not given.

put (LOCAL_FILE [, REMOTE_FILE])
: Put a file on the remote server. `LOCAL_FILE` may be a name or a filehandle. If `LOCAL_FILE` is a filehandle then `REMOTE_FILE` must be specified. If `REMOTE_FILE` is not specified then the file will be stored in the current directory with the same leafname as `LOCAL_FILE`.

  Returns `REMOTE_FILE`, or the generated remote filename if `REMOTE_FILE` is not given.

put_unique (LOCAL_FILE [, REMOTE_FILE])
: Same as put but uses the `STOU` command.

  Returns the name of the file on the server.

append (LOCAL_FILE [, REMOTE_FILE])
: Same as put but appends to the file on the remote server.

  Returns `REMOTE_FILE`, or the generated remote filename if `REMOTE_FILE` is not given.

unique_name ()
: Returns the name of the last file stored on the server using the `STOU` command.

mdtm (FILE)
: Returns the *modification time* of the given file

size (FILE)
: Returns the size in bytes for the given file.

The following methods can return different results depending on how they are called. If the user explicitly calls either of the `pasv` or `port` methods then these methods will return a *true* or *false* value. If the user does not call either of these methods then the result will be a reference to a `Net::FTP::dataconn` based object.

nlst ([DIR])
: Send a `NLST` command to the server, with an optional parameter.

list ([DIR])
: Same as `nlst` but using the `LIST` command

retr (FILE)
: Begin the retrieval of a file called `FILE` from the remote server.

stor (FILE)
: Tell the server that you wish to store a file. `FILE` is the name of the new file that should be created.

stou (FILE)
: Same as `stor` but using the `STOU` command. The name of the unique file which was created on the server will be available via the `unique_name` method after the data connection has been closed.

appe (FILE)
: Tell the server that we want to append some data to the end of a file called `FILE`. If this file does not exist then create it.

If for some reason you want to have complete control over the data connection, this includes generating it and calling the `response` method when required, then the user can use these methods to do so.

However calling these methods only affects the use of the methods above that can return a data connection. They have no effect on methods `get`, `put`, `put_unique` and those that do not require data connections.

port ([PORT])
: Send a `PORT` command to the server. If `PORT` is specified then it is sent to the server. If not the a listen socket is created and the correct information sent to the server.

pasv ()
: Tell the server to go into passive mode. Returns the text that represents the port on which the server is listening, this text is in a suitable form to sent to another ftp server using the `port` method.

The following methods can be used to transfer files between two remote servers, providing that these two servers can connect directly to each other.

pasv_xfer (SRC_FILE, DEST_SERVER [, DEST_FILE])
: This method will do a file transfer between two remote ftp servers. If `DEST_FILE` is omitted then the leaf name of `SRC_FILE` will be used.

pasv_wait (NON_PASV_SERVER)
: This method can be used to wait for a transfer to complete between a passive server and a non-passive server. The method should be called on the passive server with the `Net::FTP` object for the non-passive server passed as an argument.

abort ()
: Abort the current data transfer.

quit ()
: Send the QUIT command to the remote FTP server and close the socket connection.

Methods for the adventurous

`Net::FTP` inherits from `Net::Cmd` so methods defined in `Net::Cmd` may be used to send commands to the remote FTP server.

quot (CMD [,ARGS])
: Send a command, that Net::FTP does not directly support, to the remote server and wait for a response.

  Returns most significant digit of the response code.

  WARNING This call should only be used on commands that do not require data connections. Misuse of this method can hang the connection.

THE dataconn CLASS

Some of the methods defined in `Net::FTP` return an object which will be derived from this class.The dataconn class itself is derived from the `IO::Socket::INET` class, so any normal IO operations can be performed. However the following methods are defined in the dataconn class and IO should be performed using these.

read (BUFFER, SIZE [, TIMEOUT])
: Read `SIZE` bytes of data from the server and place it into `BUFFER`, also performing any <CRLF> translation necessary. `TIMEOUT` is optional, if not given the the timeout value from the command connection will be used.

  Returns the number of bytes read before any <CRLF> translation.

write (BUFFER, SIZE [, TIMEOUT])
: Write `SIZE` bytes of data from `BUFFER` to the server, also performing any <CRLF> translation necessary. `TIMEOUT` is optional, if not given the the timeout value from the command connection will be used.

  Returns the number of bytes written before any <CRLF> translation.

abort ()
: Abort the current data transfer.

close ()
: Close the data connection and get a response from the FTP server. Returns *true* if the connection was closed successfully and the first digit of the response from the server was a '2'.

UNIMPLEMENTED

The following RFC959 commands have not been implemented:

ALLO
: Allocates storage for the file to be transferred.

SMNT
Mount a different file system structure without changing login or accounting information.

HELP
Ask the server for "helpful information" (that's what the RFC says) on the commands it accepts.

MODE
Specifies transfer mode (stream, block or compressed) for file to be transferred.

SITE
Request remote server site parameters.

SYST
Request remote server system identification.

STAT
Request remote server status.

STRU
Specifies file structure for file to be transferred.

REIN
Reinitialize the connection, flushing all I/O and account information.

REPORTING BUGS

When reporting bugs/problems please include as much information as possible. It may be difficult for me to reproduce the problem as almost every setup is different.

A small script which yields the problem will probably be of help. It would also be useful if this script was run with the extra options `Debug = 1>` passed to the constructor, and the output sent with the bug report. If you cannot include a small script then please include a Debug trace from a run of your program which does yield the problem.

AUTHOR

Graham Barr <gbarr@ti.com>

SEE ALSO

the *Net::Netrc* manpage the *Net::Cmd* manpage

ftp (1), *ftpd* (8), RFC 959 http://www.cis.ohio-state.edu/htbin/rfc/rfc959.html

CREDITS

Henry Gabryjelski <henryg@WPI.EDU> – for the suggestion of creating directories recursively.

Nathan Torkington <gnat@frii.com> – for some input on the documentation.

Roderick Schertler <roderick@gate.net> – for various inputs

COPYRIGHT

NAME

Net::NNTP – NNTP Client class

SYNOPSIS

```
use Net::NNTP;

$nntp = Net::NNTP->new("some.host.name");
$nntp->quit;
```

DESCRIPTION

`Net::NNTP` is a class implementing a simple NNTP client in Perl as described in RFC977. `Net::NNTP` inherits its communication methods from `Net::Cmd`

CONSTRUCTOR

new ([HOST] [, OPTIONS])

This is the constructor for a new Net::NNTP object. `HOST` is the name of the remote host to which a NNTP connection is required. If not given two environment variables are checked, first `NNTPSERVER` then `NEWSHOST`, then `Net::Config` is checked, and if a host is not found then `news` is used.

`OPTIONS` are passed in a hash like fashion, using key and value pairs. Possible options are:

Timeout – Maximum time, in seconds, to wait for a response from the NNTP server, a value of zero will cause all IO operations to block. (default: 120)

Debug – Enable the printing of debugging information to STDERR

METHODS

Unless otherwise stated all methods return either a *true* or *false* value, with *true* meaning that the operation was a success. When a method states that it returns a value, failure will be returned as *undef* or an empty list.

article ([MSGID|MSGNUM])

Retrieve the header, a blank line, then the body (text) of the specified article.

If no arguments are passed then the current article in the current newsgroup is returned.

`MSGNUM` is a numeric id of an article in the current newsgroup, and will change the current article pointer. `MSGID` is the message id of an article as shown in that article's header. It is anticipated that the client will obtain the `MSGID` from a list provided by the `newnews` command, from references contained within another article, or from the message-id provided in the response to some other commands.

Returns a reference to an array containing the article.

body ([MSGID|MSGNUM])

Retrieve the body (text) of the specified article.

Takes the same arguments as `article`

Returns a reference to an array containing the body of the article.

head ([MSGID|MSGNUM])

Retrieve the header of the specified article.

Takes the same arguments as `article`

Returns a reference to an array containing the header of the article.

nntpstat ([MSGID|MSGNUM])

The `nntpstat` command is similar to the `article` command except that no text is returned. When selecting by message number within a group, the `nntpstat` command serves to set the "current article pointer" without sending text.

Using the `nntpstat` command to select by message-id is valid but of questionable value, since a selection by message-id does **not** alter the "current article pointer".

Returns the message-id of the "current article".

group ([GROUP])

Set and/or get the current group. If `GROUP` is not given then information is returned on the current group.

In a scalar context it returns the group name.

In an array context the return value is a list containing, the number of articles in the group, the number of the first article, the number of the last article and the group name.

ihave (MSGID [, MESSAGE])

The `ihave` command informs the server that the client has an article whose id is `MSGID`. If the server desires a copy of that article, and `MESSAGE` has been given the it will be sent.

Returns *true* if the server desires the article and `MESSAGE` was successfully sent,if specified.

If `MESSAGE` is not specified then the message must be sent using the `datasend` and `dataend` methods from the *Net::Cmd* manpage

`MESSAGE` can be either an array of lines or a reference to an array.

last ()

Set the "current article pointer" to the previous article in the current newsgroup.

Returns the message-id of the article.

date ()

Returns the date on the remote server. This date will be in a UNIX time format (seconds since 1970)

postok ()

`postok` will return *true* if the servers initial response indicated that it will allow posting.

authinfo (USER, PASS)

list ()

Obtain information about all the active newsgroups. The results is a reference to a hash where the key is a group name and each value is a reference to an array. The elements in this array are:– the first article number in the group, the last article number in the group and any information flags about the group.

newgroups (SINCE [, DISTRIBUTIONS])

`SINCE` is a time value and `DISTRIBUTIONS` is either a distribution pattern or a reference to a list of distribution patterns. The result is the same as `list`, but the groups return will be limited to those created after `SINCE` and, if specified, in one of the distribution areas in `DISTRIBUTIONS`.

newnews (SINCE [, GROUPS [, DISTRIBUTIONS]])

`SINCE` is a time value. `GROUPS` is either a group pattern or a reference to a list of group patterns. `DISTRIBUTIONS` is either a distribution pattern or a reference to a list of distribution patterns.

Returns a reference to a list which contains the message-ids of all news posted after `SINCE`, that are in a groups which matched `GROUPS` and a distribution which matches `DISTRIBUTIONS`.

next ()

Set the "current article pointer" to the next article in the current newsgroup.

Returns the message-id of the article.

post ([MESSAGE])

Post a new article to the news server. If `MESSAGE` is specified and posting is allowed then the message will be sent.

If `MESSAGE` is not specified then the message must be sent using the `datasend` and `dataend` methods from the *Net::Cmd* manpage

`MESSAGE` can be either an array of lines or a reference to an array.

slave ()
: Tell the remote server that I am not a user client, but probably another news server.

quit ()
: Quit the remote server and close the socket connection.

Extension methods

These methods use commands that are not part of the RFC977 documentation. Some servers may not support all of them.

newsgroups ([PATTERN])
: Returns a reference to a hash where the keys are all the group names which match `PATTERN`, or all of the groups if no pattern is specified, and each value contains the description text for the group.

distributions ()
: Returns a reference to a hash where the keys are all the possible distribution names and the values are the distribution descriptions.

subscriptions ()
: Returns a reference to a list which contains a list of groups which are recommended for a new user to subscribe to.

overview_fmt ()
: Returns a reference to an array which contain the names of the fields returned by `xover`.

active_times ()
: Returns a reference to a hash where the keys are the group names and each value is a reference to an array containing the time the groups was created and an identifier, possibly an Email address, of the creator.

active ([PATTERN])
: Similar to `list` but only active groups that match the pattern are returned. `PATTERN` can be a group pattern.

xgtitle (PATTERN)
: Returns a reference to a hash where the keys are all the group names which match `PATTERN` and each value is the description text for the group.

xhdr (HEADER, MESSAGE−SPEC)
: Obtain the header field `HEADER` for all the messages specified.

  The return value will be a reference to a hash where the keys are the message numbers and each value contains the text of the requested header for that message.

xover (MESSAGE−SPEC)
: The return value will be a reference to a hash where the keys are the message numbers and each value contains a reference to an array which contains the overview fields for that message.

  The names of the fields can be obtained by calling `overview_fmt`.

xpath (MESSAGE−ID)
: Returns the path name to the file on the server which contains the specified message.

xpat (HEADER, PATTERN, MESSAGE−SPEC)
: The result is the same as `xhdr` except the is will be restricted to headers where the text of the header matches `PATTERN`

xrover
: The XROVER command returns reference information for the *article* (s) specified.

  Returns a reference to a HASH where the keys are the message numbers and the values are the References: lines from the articles listgroup

reader

UNSUPPORTED

The following NNTP command are unsupported by the package, and there are no plans to do so.

```
AUTHINFO GENERIC
XTHREAD
XSEARCH
XINDEX
```

DEFINITIONS

MESSAGE−SPEC

`MESSAGE−SPEC` is either a single message-id, a single message number, or a reference to a list of two message numbers.

If `MESSAGE−SPEC` is a reference to a list of two message numbers and the second number in a range is less than or equal to the first then the range represents all messages in the group after the first message number.

NOTE For compatibility reasons only with earlier versions of Net::NNTP a message spec can be passed as a list of two numbers, this is depreciated and a reference to the list should now be passed

PATTERN

The `NNTP` protocol uses the `WILDMAT` format for patterns. The WILDMAT format was first developed by Rich Salz based on the format used in the UNIX ''find'' command to articulate file names. It was developed to provide a uniform mechanism for matching patterns in the same manner that the UNIX shell matches filenames.

Patterns are implicitly anchored at the beginning and end of each string when testing for a match.

There are five pattern matching operations other than a strict one-to-one match between the pattern and the source to be checked for a match.

The first is an asterisk `*` to match any sequence of zero or more characters.

The second is a question mark `?` to match any single character. The third specifies a specific set of characters.

The set is specified as a list of characters, or as a range of characters where the beginning and end of the range are separated by a minus (or dash) character, or as any combination of lists and ranges. The dash can also be included in the set as a character it if is the beginning or end of the set. This set is enclosed in square brackets. The close square bracket `]` may be used in a set if it is the first character in the set.

The fourth operation is the same as the logical not of the third operation and is specified the same way as the third with the addition of a caret character `^` at the beginning of the test string just inside the open square bracket.

The final operation uses the backslash character to invalidate the special meaning of the a open square bracket `[`, the asterisk, backslash or the question mark. Two backslashes in sequence will result in the evaluation of the backslash as a character with no special meaning.

Examples

`[^]−]` matches any single character other than a close square bracket or a minus sign/dash.

`*bdc` matches any string that ends with the string ''bdc'' including the string ''bdc'' (without quotes).

`[0−9a−zA−Z]`
matches any single printable alphanumeric ASCII character.

`a??d` matches any four character string which begins with a and ends with d.

SEE ALSO

the *Net::Cmd* manpage

AUTHOR

Graham Barr <gbarr@ti.com>

COPYRIGHT

NAME

Net::Netrc − OO interface to users netrc file

SYNOPSIS

```
use Net::Netrc;

$mach = Net::Netrc->lookup('some.machine');
$login = $mach->login;
($login, $password, $account) = $mach->lpa;
```

DESCRIPTION

`Net::Netrc` is a class implementing a simple interface to the .netrc file used as by the ftp program.

`Net::Netrc` also implements security checks just like the ftp program, these checks are, first that the .netrc file must be owned by the user and second the ownership permissions should be such that only the owner has read and write access. If these conditions are not met then a warning is output and the .netrc file is not read.

THE .netrc FILE

The .netrc file contains login and initialization information used by the auto-login process. It resides in the user's home directory. The following tokens are recognized; they may be separated by spaces, tabs, or new-lines:

machine name

Identify a remote machine name. The auto-login process searches the .netrc file for a machine token that matches the remote machine specified. Once a match is made, the subsequent .netrc tokens are processed, stopping when the end of file is reached or an- other machine or a default token is encountered.

default

This is the same as machine name except that default matches any name. There can be only one default token, and it must be after all machine tokens. This is normally used as:

```
default login anonymous password user@site
```

thereby giving the user automatic anonymous login to machines not specified in .netrc.

login name

Identify a user on the remote machine. If this token is present, the auto-login process will initiate a login using the specified name.

password string

Supply a password. If this token is present, the auto-login process will supply the specified string if the remote server requires a password as part of the login process.

account string

Supply an additional account password. If this token is present, the auto-login process will supply the specified string if the remote server requires an additional account password.

macdef name

Define a macro. `Net::Netrc` only parses this field to be compatible with *ftp*.

CONSTRUCTOR

The constructor for a `Net::Netrc` object is not called new as it does not really create a new object. But instead is called `lookup` as this is essentially what it does.

lookup (MACHINE [, LOGIN])

Lookup and return a reference to the entry for `MACHINE`. If `LOGIN` is given then the entry returned will have the given login. If `LOGIN` is not given then the first entry in the .netrc file for `MACHINE` will be returned.

If a matching entry cannot be found, and a default entry exists, then a reference to the default entry is returned.

METHODS

login ()

Return the login id for the netrc entry

password ()

Return the password for the netrc entry

account ()

Return the account information for the netrc entry

lpa ()

Return a list of login, password and account information fir the netrc entry

AUTHOR

Graham Barr <gbarr@ti.com>

SEE ALSO

the *Net::Netrc* manpage the *Net::Cmd* manpage

COPYRIGHT

NAME

Net::PH − CCSO Nameserver Client class

SYNOPSIS

```
use Net::PH;

$ph = Net::PH->new("some.host.name",
                   Port    => 105,
                   Timeout => 120,
                   Debug   => 0);

if($ph) {
    $q = $ph->query({ field1 => "value1" },
                    [qw(name address pobox)]);

    if($q) {
    }
}

# Alternative syntax

if($ph) {
    $q = $ph->query('field1=value1',
                    'name address pobox');

    if($q) {
    }
}
```

DESCRIPTION

`Net::PH` is a class implementing a simple Nameserver/PH client in Perl as described in the CCSO Nameserver -- Server-Client Protocol. Like other modules in the Net:: family the `Net::PH` object inherits methods from `Net::Cmd`.

CONSTRUCTOR

new ([HOST] [, OPTIONS])

```
$ph = Net::PH->new("some.host.name",
                   Port    => 105,
                   Timeout => 120,
                   Debug   => 0
                  );
```

This is the constructor for a new Net::PH object. `HOST` is the name of the remote host to which a PH connection is required.

If `HOST` is not given, then the `SNPP_Host` specified in `Net::Config` will be used.

`OPTIONS` is an optional list of named options which are passed in a hash like fashion, using key and value pairs. Possible options are:-

Port − Port number to connect to on remote host.

Timeout − Maximum time, in seconds, to wait for a response from the Nameserver, a value of zero will cause all IO operations to block. (default: 120)

Debug − Enable the printing of debugging information to STDERR

METHODS

Unless otherwise stated all methods return either a *true* or *false* value, with *true* meaning that the operation was a success. When a method states that it returns a value, failure will be returned as *undef* or an empty list.

query(SEARCH [, RETURN])

```
$q = $ph->query({ name => $myname },
                [qw(name email schedule)]);

foreach $handle (@{$q}) {
    foreach $field (keys %{$handle}) {
        $c = ${$handle}{$field}->code;
        $v = ${$handle}{$field}->value;
        $f = ${$handle}{$field}->field;
        $t = ${$handle}{$field}->text;
        print "field:[$field] [$c][$v][$f][$t]\n" ;
    }
}
```

Search the database and return fields from all matching entries.

The `SEARCH` argument is a reference to a HASH which contains field/value pairs which will be passed to the Nameserver as the search criteria.

`RETURN` is optional, but if given it should be a reference to a list which contains field names to be returned.

The alternative syntax is to pass strings instead of references, for example

```
$q = $ph->query('name=myname',
                'name email schedule');
```

The `SEARCH` argument is a string that is passed to the Nameserver as the search criteria.

`RETURN` is optional, but if given it should be a string which will contain field names to be returned.

Each match from the server will be returned as a HASH where the keys are the field names and the values are `Net::PH:Result` objects (*code*, *value*, *field*, *text*).

Returns a reference to an ARRAY which contains references to HASHs, one per match from the server.

change(SEARCH , MAKE)

```
$r = $ph->change({ email => "*.domain.name" },
                 { schedule => "busy");
```

Change field values for matching entries.

The `SEARCH` argument is a reference to a HASH which contains field/value pairs which will be passed to the Nameserver as the search criteria.

The `MAKE` argument is a reference to a HASH which contains field/value pairs which will be passed to the Nameserver that will set new values to designated fields.

The alternative syntax is to pass strings instead of references, for example

```
$r = $ph->change('email="*.domain.name"',
                 'schedule="busy"');
```

The `SEARCH` argument is a string to be passed to the Nameserver as the search criteria.

The `MAKE` argument is a string to be passed to the Nameserver that will set new values to designated fields.

Upon success all entries that match the search criteria will have the field values, given in the Make argument, changed.

login(USER, PASS [, ENCRYPT])

```
$r = $ph->login('username','password',1);
```

Enter login mode using `USER` and `PASS`. If `ENCRYPT` is given and is *true* then the password will be used to encrypt a challenge text string provided by the server, and the encrypted string will be sent back to the server. If `ENCRYPT` is not given, or *false* the the password will be sent in clear text (*this is not recommended*)

logout()

```
$r = $ph->logout();
```

Exit login mode and return to anonymous mode.

fields([FIELD_LIST])

```
$fields = $ph->fields();
foreach $field (keys %{$fields}) {
    $c = ${$fields}{$field}->code;
    $v = ${$fields}{$field}->value;
    $f = ${$fields}{$field}->field;
    $t = ${$fields}{$field}->text;
    print "field:[$field] [$c][$v][$f][$t]\n";
}
```

Returns a reference to a HASH. The keys of the HASH are the field names and the values are `Net::PH:Result` objects (*code*, *value*, *field*, *text*).

`FIELD_LIST` is a string that lists the fields for which info will be returned.

add(FIELD_VALUES)

```
$r = $ph->add( { name => $name, phone => $phone });
```

This method is used to add new entries to the Nameserver database. You must successfully call the *login* manpage before this method can be used.

Note that this method adds new entries to the database. To modify an existing entry use the *change* manpage.

`FIELD_VALUES` is a reference to a HASH which contains field/value pairs which will be passed to the Nameserver and will be used to initialize the new entry.

The alternative syntax is to pass a string instead of a reference, for example

```
$r = $ph->add('name=myname phone=myphone');
```

`FIELD_VALUES` is a string that consists of field/value pairs which the new entry will contain.

delete(FIELD_VALUES)

```
$r = $ph->delete('name=myname phone=myphone');
```

This method is used to delete existing entries from the Nameserver database. You must successfully call the *login* manpage before this method can be used.

Note that this method deletes entries to the database. To modify an existing entry use the *change* manpage.

`FIELD_VALUES` is a string that serves as the search criteria for the records to be deleted. Any entry in the database which matches this search criteria will be deleted.

id([ID])

```
$r = $ph->id('709');
```

Sends `ID` to the Nameserver, which will enter this into its logs. If `ID` is not given then the UID of the user running the process will be sent.

status()

Returns the current status of the Nameserver.

siteinfo()

```
$siteinfo = $ph->siteinfo();
foreach $field (keys %{$siteinfo}) {
    $c = ${$siteinfo}{$field}->code;
    $v = ${$siteinfo}{$field}->value;
    $f = ${$siteinfo}{$field}->field;
    $t = ${$siteinfo}{$field}->text;
    print "field:[$field] [$c][$v][$f][$t]\n";
}
```

Returns a reference to a HASH containing information about the server's site. The keys of the HASH are the field names and values are `Net::PH:Result` objects (*code*, *value*, *field*, *text*).

quit()

```
$r = $ph->quit();
```

Quit the connection

Q&A

How do I get the values of a Net::PH::Result object?

```
foreach $handle (@{$q}) {
    foreach $field (keys %{$handle}) {
        $my_code  = ${$q}{$field}->code;
        $my_value = ${$q}{$field}->value;
        $my_field = ${$q}{$field}->field;
        $my_text  = ${$q}{$field}->text;
    }
}
```

How do I get a count of the returned matches to my query?

```
$my_count = scalar(@{$query_result});
```

How do I get the status code and message of the last `$ph` command?

```
$status_code    = $ph->code;
$status_message = $ph->message;
```

SEE ALSO

the *Net::Cmd* manpage

AUTHORS

Graham Barr <gbarr@ti.com> Alex Hristov <hristov@slb.com>

ACKNOWLEDGMENTS

Password encryption code ported to perl by Broc Seib <bseib@purdue.edu>, Purdue University Computing Center.

Otis Gospodnetic <otisg@panther.middlebury.edu> suggested passing parameters as string constants. Some queries cannot be executed when passing parameters as string references.

```
Example: query first_name last_name email="*.domain"
```

COPYRIGHT

NAME

Net::POP3 – Post Office Protocol 3 Client class (RFC1081)

SYNOPSIS

```
use Net::POP3;

# Constructors
$pop = Net::POP3->new('pop3host');
$pop = Net::POP3->new('pop3host', Timeout => 60);
```

DESCRIPTION

This module implements a client interface to the POP3 protocol, enabling a perl5 application to talk to POP3 servers. This documentation assumes that you are familiar with the POP3 protocol described in RFC1081.

A new Net::POP3 object must be created with the *new* method. Once this has been done, all POP3 commands are accessed via method calls on the object.

EXAMPLES

```
Need some small examples in here :-)
```

CONSTRUCTOR

new ([HOST,] [OPTIONS])
: This is the constructor for a new Net::POP3 object. `HOST` is the name of the remote host to which a POP3 connection is required.

  If `HOST` is not given, then the `POP3_Host` specified in `Net::Config` will be used.

  `OPTIONS` are passed in a hash like fashion, using key and value pairs. Possible options are:

  Timeout – Maximum time, in seconds, to wait for a response from the POP3 server (default: 120)

  Debug – Enable debugging information

METHODS

Unless otherwise stated all methods return either a *true* or *false* value, with *true* meaning that the operation was a success. When a method states that it returns a value, failure will be returned as *undef* or an empty list.

user (USER)
: Send the USER command.

pass (PASS)
: Send the PASS command. Returns the number of messages in the mailbox.

login ([USER [, PASS]])
: Send both the the USER and PASS commands. If `PASS` is not given the `Net::POP3` uses `Net::Netrc` to lookup the password using the host and username. If the username is not specified then the current user name will be used.

  Returns the number of messages in the mailbox.

top (MSGNUM [, NUMLINES])
: Get the header and the first `NUMLINES` of the body for the message `MSGNUM`. Returns a reference to an array which contains the lines of text read from the server.

list ([MSGNUM])
: If called with an argument the `list` returns the size of the message in octets.

  If called without arguments a reference to a hash is returned. The keys will be the `MSGNUM`'s of all undeleted messages and the values will be their size in octets.

get (MSGNUM)

Get the message `MSGNUM` from the remote mailbox. Returns a reference to an array which contains the lines of text read from the server.

last ()

Returns the highest `MSGNUM` of all the messages accessed.

popstat ()

Returns an array of two elements. These are the number of undeleted elements and the size of the mbox in octets.

delete (MSGNUM)

Mark message `MSGNUM` to be deleted from the remote mailbox. All messages that are marked to be deleted will be removed from the remote mailbox when the server connection closed.

reset ()

Reset the status of the remote POP3 server. This includes reseting the status of all messages to not be deleted.

quit ()

Quit and close the connection to the remote POP3 server. Any messages marked as deleted will be deleted from the remote mailbox.

NOTES

If a `Net::POP3` object goes out of scope before `quit` method is called then the `reset` method will called before the connection is closed. This means that any messages marked to be deleted will not be.

SEE ALSO

the *Net::Netrc* manpage the *Net::Cmd* manpage

AUTHOR

Graham Barr <gbarr@ti.com>

COPYRIGHT

NAME

Net::SMTP − Simple Mail Transfer Protocol Client

SYNOPSIS

```
use Net::SMTP;

# Constructors
$smtp = Net::SMTP->new('mailhost');
$smtp = Net::SMTP->new('mailhost', Timeout => 60);
```

DESCRIPTION

This module implements a client interface to the SMTP and ESMTP protocol, enabling a perl5 application to talk to SMTP servers. This documentation assumes that you are familiar with the concepts of the SMTP protocol described in RFC821.

A new Net::SMTP object must be created with the *new* method. Once this has been done, all SMTP commands are accessed through this object.

The Net::SMTP class is a subclass of Net::Cmd and IO::Socket::INET.

EXAMPLES

This example prints the mail domain name of the SMTP server known as mailhost:

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

use Net::SMTP;

$smtp = Net::SMTP->new('mailhost');
print $smtp->domain,"\n";
$smtp->quit;
```

This example sends a small message to the postmaster at the SMTP server known as mailhost:

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

use Net::SMTP;

$smtp = Net::SMTP->new('mailhost');

$smtp->mail($ENV{USER});
$smtp->to('postmaster');

$smtp->data();
$smtp->datasend("To: postmaster\n");
$smtp->datasend("\n");
$smtp->datasend("A simple test message\n");
$smtp->dataend();

$smtp->quit;
```

CONSTRUCTOR

new Net::SMTP [HOST,] [OPTIONS]

This is the constructor for a new Net::SMTP object. `HOST` is the name of the remote host to which a SMTP connection is required.

If `HOST` is not given, then the `SMTP_Host` specified in `Net::Config` will be used.

`OPTIONS` are passed in a hash like fashion, using key and value pairs. Possible options are:

Hello – SMTP requires that you identify yourself. This option specifies a string to pass as your mail domain. If not given a guess will be taken.

Timeout – Maximum time, in seconds, to wait for a response from the SMTP server (default: 120)

Debug – Enable debugging information

Example:

```
$smtp = Net::SMTP->new('mailhost',
                        Hello => 'my.mail.domain'
                        Timeout => 30,
                        Debug   => 1,
                       );
```

METHODS

Unless otherwise stated all methods return either a *true* or *false* value, with *true* meaning that the operation was a success. When a method states that it returns a value, failure will be returned as *undef* or an empty list.

domain ()
: Returns the domain that the remote SMTP server identified itself as during connection.

hello (DOMAIN)
: Tell the remote server the mail domain which you are in using the EHLO command (or HELO if EHLO fails). Since this method is invoked automatically when the Net::SMTP object is constructed the user should normally not have to call it manually.

mail (ADDRESS [, OPTIONS])

send (ADDRESS)

send_or_mail (ADDRESS)

send_and_mail (ADDRESS)
: Send the appropriate command to the server MAIL, SEND, SOML or SAML. `ADDRESS` is the address of the sender. This initiates the sending of a message. The method `recipient` should be called for each address that the message is to be sent to.

The `mail` method can some additional ESMTP OPTIONS which is passed in hash like fashion, using key and value pairs. Possible options are:

```
Size        => <bytes>
Return      => <???>
Bits        => "7" | "8"
Transaction => <ADDRESS>
Envelope    => <ENVID>
```

reset ()
: Reset the status of the server. This may be called after a message has been initiated, but before any data has been sent, to cancel the sending of the message.

recipient (ADDRESS [, ADDRESS [...]])
: Notify the server that the current message should be sent to all of the addresses given. Each address is sent as a separate command to the server. Should the sending of any address result in a failure then the process is aborted and a *false* value is returned. It is up to the user to call `reset` if they so desire.

to (ADDRESS [, ADDRESS [...]])
: A synonym for `recipient`.

data ([DATA])
: Initiate the sending of the data from the current message.

  `DATA` may be a reference to a list or a list. If specified the contents of `DATA` and a termination string `".\r\n"` is sent to the server. And the result will be true if the data was accepted.

  If `DATA` is not specified then the result will indicate that the server wishes the data to be sent. The data must then be sent using the `datasend` and `dataend` methods described in the *Net::Cmd* manpage.

expand (ADDRESS)
: Request the server to expand the given address Returns a reference to an array which contains the text read from the server.

verify (ADDRESS)
: Verify that `ADDRESS` is a legitimate mailing address.

help ([$subject])
: Request help text from the server. Returns the text or undef upon failure

quit ()
: Send the QUIT command to the remote SMTP server and close the socket connection.

SEE ALSO

the *Net::Cmd* manpage

AUTHOR

Graham Barr <gbarr@ti.com>

COPYRIGHT

NAME

Net::SNPP – Simple Network Pager Protocol Client

SYNOPSIS

```
use Net::SNPP;

# Constructors
$snpp = Net::SNPP->new('snpphost');
$snpp = Net::SNPP->new('snpphost', Timeout => 60);
```

NOTE

This module is not complete, yet !

DESCRIPTION

This module implements a client interface to the SNPP protocol, enabling a perl5 application to talk to SNPP servers. This documentation assumes that you are familiar with the SNPP protocol described in RFC1861.

A new Net::SNPP object must be created with the *new* method. Once this has been done, all SNPP commands are accessed through this object.

EXAMPLES

This example will send a pager message in one hour saying "Your lunch is ready"

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

use Net::SNPP;

$snpp = Net::SNPP->new('snpphost');

$snpp->send( Pager   => $some_pager_number,
             Message => "Your lunch is ready",
             Alert   => 1,
             Hold    => time + 3600, # lunch ready in 1 hour :-)
           ) || die $snpp->message;

$snpp->quit;
```

CONSTRUCTOR

new ([HOST,] [OPTIONS])

This is the constructor for a new Net::SNPP object. `HOST` is the name of the remote host to which a SNPP connection is required.

If `HOST` is not given, then the `SNPP_Host` specified in `Net::Config` will be used.

`OPTIONS` are passed in a hash like fashion, using key and value pairs. Possible options are:

Timeout – Maximum time, in seconds, to wait for a response from the SNPP server (default: 120)

Debug – Enable debugging information

Example:

```
$snpp = Net::SNPP->new('snpphost',
                       Debug => 1,
                      );
```

METHODS

Unless otherwise stated all methods return either a *true* or *false* value, with *true* meaning that the operation was a success. When a method states that it returns a value, failure will be returned as *undef* or an empty list.

reset ()

help () Request help text from the server. Returns the text or undef upon failure

quit () Send the QUIT command to the remote SNPP server and close the socket connection.

EXPORTS

`Net::SNPP` exports all that `Net::CMD` exports, plus three more subroutines that can bu used to compare against the result of `status`. These are :– `CMD_2WAYERROR`, `CMD_2WAYOK`, and `CMD_2WAYQUEUED`.

SEE ALSO

the *Net::Cmd* manpage RFC1861

AUTHOR

Graham Barr <gbarr@ti.com>

COPYRIGHT

NAME

Net::Telnet − Defines constants for the telnet protocol

SYNOPSIS

```
use Telnet qw(TELNET_IAC TELNET_DO TELNET_DONT);
```

DESCRIPTION

This module is **VERY** preliminary as I am not 100% sure how it should be implemented.

Currently it just exports constants used in the telnet protocol.

Should it contain sub's for packing and unpacking commands ?

Please feel free to send me any suggestions

NOTE

This is not an implementation of the 'telnet' command but of the telnet protocol as defined in RFC854

AUTHOR

Graham Barr <gbarr@ti.com>

COPYRIGHT

NAME

Net::Time − time and daytime network client interface

SYNOPSIS

```
use Net::Time qw(inet_time inet_daytime);

print inet_time();          # use default host from Net::Config
print inet_time('localhost');
print inet_time('localhost', 'tcp');

print inet_daytime();       # use default host from Net::Config
print inet_daytime('localhost');
print inet_daytime('localhost', 'tcp');
```

DESCRIPTION

`Net::Time` provides subroutines that obtain the time on a remote machine.

inet_time ([HOST [, PROTOCOL [, TIMEOUT]]])

Obtain the time on `HOST`, or some default host if `HOST` is not given or not defined, using the protocol as defined in RFC868. The optional argument `PROTOCOL` should define the protocol to use, either `tcp` or `udp`. The result will be a unix-like time value or *undef* upon failure.

inet_daytime ([HOST [, PROTOCOL [, TIMEOUT]]])

Obtain the time on `HOST`, or some default host if `HOST` is not given or not defined, using the protocol as defined in RFC867. The optional argument `PROTOCOL` should define the protocol to use, either `tcp` or `udp`. The result will be an ASCII string or *undef* upon failure.

AUTHOR

Graham Barr <gbarr@ti.com>

COPYRIGHT