NAME

LWP − The World−Wide Web library for Perl

SYNOPSIS

```
  use LWP;
  print "This is libwww−perl−$LWP::VERSION\n";
```

DESCRIPTION

The libwww-perl collection is a set of Perl modules which provides a simple and consistent application 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. The library also contain modules that are of more general use and even classes that help you implement simple HTTP servers.

Most modules in this library provide an object oriented API. 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 is easy to extend and customize for your own 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`, `https`, `gopher`, `ftp`, `news`, `file`, and `mailto` resources.
- Provides a full object oriented interface or a very simple procedural interface.
- Supports the basic and digest authorization schemes.
- Supports transparent redirect handling.
- Supports access through proxy servers.
- Provides parser for *robots.txt* files and a framework for constructing robots.
- Supports parsing of HTML forms.
- Implements HTTP content negotiation algorithm that can be used both in protocol modules and in server scripts (like CGI scripts).
- Supports HTTP cookies.
- Some simple command line clients, for instance `lwp−request` and `lwp−download`.

HTTP STYLE COMMUNICATION

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

Let us start with this quote from the HTTP specification document <http://www.w3.org/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 libwww-perl request object has the class name HTTP::Request. The fact that the class name uses `HTTP::` as a 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:

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

The Response Object

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

- **code** is a numerical value that indicates the overall outcome of the request.
- **message** is a short, human readable string that corresponds to the *code*.
- **headers** contains additional information about the response and describe the content.
- **content** is an arbitrary amount of data.

Since we don’t want to handle all possible *code* values directly in our programs, a libwww-perl response object has 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 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 to a *user agent* object and this object takes care of all the things that need to be done (like low-level communication and error handling) and returns a *response* object. The user agent represents your application on the network and provides you with an interface that can accept *requests* and return *responses*.

The user agent is 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 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 class. The main method provided by this object is **request()**. This method takes an HTTP::Request object as argument and (eventually) returns a HTTP::Response object.

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

- **timeout** specifies how much time we give remote servers to respond before the library disconnects and creates an internal *timeout* response.
- **agent** specifies the name that your application uses when it presents itself on the network.
- **from** 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.
- **parse_head** specifies whether we should initialize response headers from the `<head>` section of HTML documents.
- **proxy** and **no_proxy** specify if and when to go through a proxy server. <http://www.w3.org/History/1994/WWW/Proxies/>
- **credentials** provides a way to set up user names and passwords needed to access certain services.

Many applications want even more control over how they interact with the network and they get this by sub-classing LWP::UserAgent. The library includes a sub-class, LWP::RobotUA, for 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;
my $ua = LWP::UserAgent−>new;
$ua−>agent("MyApp/0.1 ");

# Create a request
my $req = HTTP::Request−>new(POST => 'http://search.cpan.org/search');
$req−>content_type('application/x−www−form−urlencoded');
$req−>content('query=libwww−perl&mode=dist');

# 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 $res−>status_line, "\n";
}
```

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

NETWORK SUPPORT

This section discusses the various protocol schemes and the HTTP style methods that headers may be used for each.

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

For all responses, the library adds a header called ''Client-Date''. This header holds the time when the response was received by your application. The format and semantics of the header are the same as the server created ''Date'' header. You may also encounter other ''Client-XXX'' headers. They are all generated by the library internally and are not received from the servers.

HTTP Requests

HTTP requests are just handed off to an HTTP server and it decides what happens. Few servers implement methods beside the usual ''GET'', ''HEAD'', ''POST'' and ''PUT'', but CGI-scripts may 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 a GET request you might want to add an ‘‘If-Modified-Since’’ or ‘‘If-None-Match’’ header to make the request conditional.

For a 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 lwpcook for examples of this.

The libwww-perl HTTP implementation currently support the HTTP/1.1 and HTTP/1.0 protocol.

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 LWP::UserAgent for documentation of this.

HTTPS Requests

HTTPS requests are HTTP requests over an encrypted network connection using the SSL protocol developed by Netscape. Everything about HTTP requests above also apply to HTTPS requests. In addition the library will add the headers ‘‘Client-SSL-Cipher’’, ‘‘Client-SSL-Cert-Subject’’ and ‘‘Client-SSL-Cert-Issuer’’ to the response. These headers denote the encryption method used and the name of the server owner.

The request can contain the header ‘‘If-SSL-Cert-Subject’’ in order to make the request conditional on the content of the server certificate. If the certificate subject does not match, no request is sent to the server and an internally generated error response is returned. The value of the ‘‘If-SSL-Cert-Subject’’ header is interpreted as a Perl regular expression.

FTP Requests

The library currently supports GET, HEAD and PUT requests. GET retrieves a file or a directory listing from an FTP server. PUT stores a file on a ftp server.

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

User name/password can be specified using basic authorization or be encoded in the URL. Failed logins return an UNAUTHORIZED response with ‘‘WWW-Authenticate: Basic’’ and can be treated like basic authorization for HTTP.

The library supports ftp ASCII transfer mode by specifying the ‘‘type=a’’ parameter in the URL. It also supports transfer of ranges for FTP transfers using the ‘‘Range’’ header.

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 provides methods 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).

For normal file retrievals, the ‘‘Content-Type’’ is guessed based on the file name suffix. See LWP::MediaTypes.

The ‘‘If-Modified-Since’’ request header works for servers that implement the `MDTM` command. It will probably not work for directory listings though.

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 news: URLs.

The library supports 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 methods for gopher requests. All request header values are ignored. HEAD cheats and returns 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.org');
$req->header(Subject => "subscribe");
$req->content("Please subscribe me to the libwww-perl mailing list!\n");
```

CPAN Requests

URLs with scheme `cpan:` are redirected to a suitable CPAN mirror. If you have your own local mirror of CPAN you might tell LWP to use it for `cpan:` URLs by an assignment like this:

```
$LWP::Protocol::cpan::CPAN = "file:/local/CPAN/";
```

Suitable CPAN mirrors are also picked up from the configuration for the CPAN.pm, so if you have used that module a suitable mirror should be picked automatically. If neither of these apply, then a redirect to the generic CPAN http location is issued.

Example request to download the newest perl:

```
 $req = HTTP::Request->new(GET => "cpan:src/latest.tar.gz");
```

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::Authen::Basic -- Handle 401 and 407 responses
 LWP::Authen::Digest

 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

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

 Net::HTTP          -- Low level HTTP client
```

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::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
 File::Listing      -- Parse directory listings
 HTML::Form         -- Processing for <form>s in HTML documents
```

MORE DOCUMENTATION

All modules contain detailed information on the interfaces they provide. 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-download, lwp-dump and lwp-mirror are implemented.

ENVIRONMENT

The following environment variables are used by LWP:

HOME

The LWP::MediaTypes functions will look for the *.media.types* and *.mime.types* files relative to you home directory.

http_proxy
ftp_proxy
xxx_proxy
no_proxy

These environment variables can be set to enable communication through a proxy server. See the description of the `env_proxy` method in LWP::UserAgent.

PERL_LWP_ENV_PROXY

If set to a TRUE value, then the LWP::UserAgent will by default call `env_proxy` during initialization. This makes LWP honor the proxy variables described above.

PERL_LWP_SSL_VERIFY_HOSTNAME

The default `verify_hostname` setting for LWP::UserAgent. If not set the default will be 1. Set it as 0 to disable hostname verification (the default prior to libwww-perl 5.840.

PERL_LWP_SSL_CA_FILE
PERL_LWP_SSL_CA_PATH

The file and/or directory where the trusted Certificate Authority certificates is located. See LWP::UserAgent for details.

PERL_HTTP_URI_CLASS

Used to decide what URI objects to instantiate. The default is URI. You might want to set it to URI::URL for compatibility with old times.

AUTHORS

LWP was made possible by contributions from Adam Newby, Albert Dvornik, Alexandre Duret-Lutz, Andreas Gustafsson, Andreas König, Andrew Pimlott, Andy Lester, Ben Coleman, Benjamin Low, Ben Low, Ben Tilly, Blair Zajac, Bob Dalgleish, BooK, Brad Hughes, Brian J. Murrell, Brian McCauley, Charles C. Fu, Charles Lane, Chris Nandor, Christian Gilmore, Chris W. Unger, Craig Macdonald, Dale Couch, Dan Kubb, Dave Dunkin, Dave W. Smith, David Coppit, David Dick, David D. Kilzer, Doug MacEachern, Edward Avis, erik, Gary Shea, Gisle Aas, Graham Barr, Gurusamy Sarathy, Hans de Graaff, Harald Joerg, Harry Bochner, Hugo, Ilya Zakharevich, INOUE Yoshinari, Ivan Panchenko, Jack Shirazi, James Tillman, Jan Dubois, Jared Rhine, Jim Stern, Joao Lopes, John Klar, Johnny Lee, Josh Kronengold, Josh Rai, Joshua Chamas, Joshua Hoblitt, Kartik Subbarao, Keiichiro Nagano, Ken Williams, KONISHI Katsuhiro, Lee T Lindley, Liam Quinn, Marc Hedlund, Marc Langheinrich, Mark D. Anderson, Marko Asplund, Mark Stosberg, Markus B Krüger, Markus Laker, Martijn Koster, Martin Thurn, Matthew Eldridge, Matthew.van.Eerde, Matt Sergeant, Michael A. Chase, Michael Quaranta, Michael Thompson, Mike Schilli, Moshe Kaminsky, Nathan Torkington, Nicolai Langfeldt, Norton Allen, Olly Betts, Paul J. Schinder, peterm, Philip Guenther, Daniel Buenzli, Pon Hwa Lin, Radoslaw Zielinski, Radu Greab, Randal L. Schwartz, Richard Chen, Robin Barker, Roy Fielding, Sander van Zoest, Sean M. Burke, shildreth, Slaven Rezic, Steve A Fink, Steve Hay, Steven Butler, Steve_Kilbane, Takanori Ugai, Thomas Lotterer, Tim Bunce, Tom Hughes, Tony Finch, Ville Skyttä, Ward Vandewege, William York, Yale Huang, and Yitzchak Scott-Thoennes.

LWP owes a lot in motivation, design, and code, to the libwww-perl library for Perl4 by Roy Fielding, which included 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.

COPYRIGHT

```
Copyright 1995−2009, Gisle Aas
Copyright 1995, Martijn Koster
```

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

AVAILABILITY

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

```
http://github.com/libwww−perl/libwww−perl
```

The best place to discuss this code is on the <libwww@perl.org> mailing list.