NAME

LWP::UserAgent - Web user agent class

SYNOPSIS

```
require LWP::UserAgent;

my $ua = LWP::UserAgent->new;
$ua->timeout(10);
$ua->env_proxy;

my $response = $ua->get('http://search.cpan.org/');

if ($response->is_success) {
print $response->decoded_content; # or whatever
}
else {
die $response->status_line;
}
```

DESCRIPTION

The `LWP::UserAgent` is a class implementing a web user agent. `LWP::UserAgent` objects can be used to dispatch web requests.

In normal use the application creates an `LWP::UserAgent` object, and then configures it with values for timeouts, proxies, name, etc. It then creates an instance of `HTTP::Request` for the request that needs to be performed. This request is then passed to one of the request method the UserAgent, which dispatches it using the relevant protocol, and returns a `HTTP::Response` object. There are convenience methods for sending the most common request types: *get()*, *head()*, *post()*, *put()* and *delete()*. When using these methods then the creation of the request object is hidden as shown in the synopsis above.

The basic approach of the library is to use HTTP style communication for all protocol schemes. This means that you will construct `HTTP::Request` objects and receive `HTTP::Response` objects even for non-HTTP resources like *gopher* and *ftp*. In order to achieve even more similarity to HTTP style communications, gopher menus and file directories are converted to HTML documents.

CONSTRUCTOR METHODS

The following constructor methods are available:

$ua = LWP::UserAgent->new(`%options`)

This method constructs a new `LWP::UserAgent` object and returns it. Key/value pair arguments may be provided to set up the initial state. The following options correspond to attribute methods described below:

```
KEY DEFAULT
----------- --------------------
agent "libwww-perl/#.###"
from undef
conn_cache undef
cookie_jar undef
default_headers HTTP::Headers->new
local_address undef
ssl_opts { verify_hostname => 1 }
max_size undef
max_redirect 7
parse_head 1
protocols_allowed undef
protocols_forbidden undef
requests_redirectable ['GET', 'HEAD']
```

```
timeout 180
```

The following additional options are also accepted: If the `env_proxy` option is passed in with a TRUE value, then proxy settings are read from environment variables (see *env_proxy()* method below). If `env_proxy` isn't provided the `PERL_LWP_ENV_PROXY` environment variable controls if *env_proxy()* is called during initialization. If the `keep_alive` option is passed in, then a `LWP::ConnCache` is set up (see *conn_cache()* method below). The `keep_alive` value is passed on as the `total_capacity` for the connection cache.

$ua->clone
Returns a copy of the LWP::UserAgent object.

ATTRIBUTES

The settings of the configuration attributes modify the behaviour of the `LWP::UserAgent` when it dispatches requests. Most of these can also be initialized by options passed to the constructor method.

The following attribute methods are provided. The attribute value is left unchanged if no argument is given. The return value from each method is the old attribute value.

$ua->agent
$ua->agent(`$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 is the string returned by the *_agent()* method (see below).

If the `$product_id` ends with space then the *_agent()* string is appended to it.

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('Checkbot/0.4 '); # same as above
$ua->agent('Mozilla/5.0');
$ua->agent(""); # don't identify
```

$ua->_agent
Returns the default agent identifier. This is a string of the form "libwww-perl/#.###", where "#.###" is substituted with the version number of this library.

$ua->from
$ua->from(`$email_address`)
Get/set the 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` v alue is send as the "From" header in the requests. Example:

```
$ua->from('gaas@cpan.org');
```

The default is to not send a "From" header. See the *default_headers()* method for the more general interface that allow any header to be defaulted.

$ua->cookie_jar
$ua->cookie_jar(`$cookie_jar_obj`)
Get/set the cookie jar object to use. The only requirement is that the cookie jar object must implement the extract_cookies($response) and add_cookie_header($request) methods. These methods will then be invoked by the user agent as requests are sent and responses are received. Normally this will be a `HTTP::Cookies` object or some subclass.

The default is to have no cookie_jar, i.e. never automatically add "Cookie" headers to the requests.

Shortcut: If a reference to a plain hash is passed in as the `$cookie_jar_object`, then it is replaced with an instance of `HTTP::Cookies` that is initialized based on the hash. This form also automatically loads the `HTTP::Cookies` module. It means that:

```
$ua->cookie_jar({ file => "$ENV{HOME}/.cookies.txt" });
```

is really just a shortcut for:

```
require HTTP::Cookies;
$ua->cookie_jar(HTTP::Cookies->new(file => "$ENV{HOME}/.cookies.txt"));
```

$ua->default_headers
$ua->default_headers(`$headers_obj`)
Get/set the headers object that will provide default header values for any requests sent. By default this will be an empty `HTTP::Headers` object.

$ua->default_header(`$field`)
$ua->default_header(`$field` => `$value`)
This is just a short-cut for $ua->default_headers->header(`$field` => `$value`). Example:

```
$ua->default_header('Accept-Encoding' => scalar HTTP::Message::decodable());
$ua->default_header('Accept-Language' => "no, en");
```

$ua->conn_cache
$ua->conn_cache(`$cache_obj`)
Get/set the `LWP::ConnCache` object to use. See LWP::ConnCache for details.

$ua->credentials(`$netloc`, `$realm`)
$ua->credentials(`$netloc`, `$realm`, `$uname`, `$pass`)
Get/set the user name and password to be used for a realm.

The `$netloc` is a string of the form "<host>:<port>". The username and password will only be passed to this server. Example:

```
$ua->credentials("www.example.com:80", "Some Realm", "foo", "secret");
```

$ua->local_address
$ua->local_address(`$address`)
Get/set the local interface to bind to for network connections. The interface can be specified as a hostname or an IP address. This value is passed as the `LocalAddr` argument to IO::Socket::INET.

$ua->max_size
$ua->max_size(`$bytes`)
Get/set the size limit for response content. The default is `undef`, which means that there is no limit. If the returned response content is only partial, because the size limit was exceeded, then a "Client-Aborted" header will be added to the response. The content might end up longer than `max_size` as we abort once appending a chunk of data makes the length exceed the limit. The "Content-Length" header, if present, will indicate the length of the full content and will normally not be the same as `length($res->content)`.

$ua->max_redirect
$ua->max_redirect(`$n`)
This reads or sets the object's limit of how many times it will obey redirection responses in a given request cycle.

By default, the value is 7. This means that if you call *request()* method and the response is a redirect elsewhere which is in turn a redirect, and so on seven times, then LWP gives up after that seventh request.

$ua->parse_head
$ua->parse_head(`$boolean`)
Get/set a value indicating whether 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->protocols_allowed
$ua->protocols_allowed(@protocols)

This reads (or sets) this user agent's list of protocols that the request methods will exclusively allow. The protocol names are case insensitive.

For example: `$ua->protocols_allowed( [ 'http', 'https'] );` means that this user agent will *allow only* those protocols, and attempts to use this user agent to access URLs with any other schemes (like "ftp://...") will result in a 500 error.

To delete the list, call: `$ua->protocols_allowed(undef)`

By default, an object has neither a `protocols_allowed` list, nor a `protocols_forbidden` list.

Note that having a `protocols_allowed` list causes any `protocols_forbidden` list to be ignored.

$ua->protocols_forbidden
$ua->protocols_forbidden(@protocols)

This reads (or sets) this user agent's list of protocols that the request method will *not* allow. The protocol names are case insensitive.

For example: `$ua->protocols_forbidden( [ 'file', 'mailto'] );` means that this user agent will *not* allow those protocols, and attempts to use this user agent to access URLs with those schemes will result in a 500 error.

To delete the list, call: `$ua->protocols_forbidden(undef)`

$ua->requests_redirectable
$ua->requests_redirectable(@requests)

This reads or sets the object's list of request names that `$ua->redirect_ok(...)` will allow redirection for. By default, this is `['GET', 'HEAD']`, as per RFC 2616. To change to include 'POST', consider:

```
 push @{ $ua->requests_redirectable }, 'POST';
```

$ua->show_progress
$ua->show_progress($boolean)

Get/set a value indicating whether a progress bar should be displayed on the terminal as requests are processed. The default is FALSE.

$ua->timeout
$ua->timeout($secs)

Get/set the timeout value in seconds. The default *timeout()* value is 180 seconds, i.e. 3 minutes.

The requests is aborted if no activity on the connection to the server is observed for `timeout` seconds. This means that the time it takes for the complete transaction and the *request()* method to actually return might be longer.

$ua->ssl_opts
$ua->ssl_opts($key)
$ua->ssl_opts($key => $value)

Get/set the options for SSL connections. Without argument return the list of options keys currently set. With a single argument return the current value for the given option. With 2 arguments set the option value and return the old. Setting an option to the value `undef` removes this option.

The options that LWP relates to are:

`verify_hostname => $bool`

When TRUE LWP will for secure protocol schemes ensure it connects to servers that have a valid certificate matching the expected hostname. If FALSE no checks are made and you can't be sure that you communicate with the expected peer. The no checks behaviour was the default for libwww-perl-5.837 and earlier releases.

This option is initialized from the PERL_LWP_SSL_VERIFY_HOSTNAME environment variable. If this environment variable isn't set; then `verify_hostname` defaults to 1.

`SSL_ca_file` => `$path`
: The path to a file containing Certificate Authority certificates. A default setting for this option is provided by checking the environment variables `PERL_LWP_SSL_CA_FILE` and `HTTPS_CA_FILE` in order.

`SSL_ca_path` => `$path`
: The path to a directory containing files containing Certificate Authority certificates. A default setting for this option is provided by checking the environment variables `PERL_LWP_SSL_CA_PATH` and `HTTPS_CA_DIR` in order.

Other options can be set and are processed directly by the SSL Socket implementation in use. See IO::Socket::SSL or Net::SSL for details.

The libwww-perl core no longer bundles protocol plugins for SSL. You will need to install LWP::Protocol::https separately to enable support for processing https-URLs.

Proxy attributes

The following methods set up when requests should be passed via a proxy server.

$ua->proxy(@schemes, `$proxy_url`)
$ua->proxy($scheme, `$proxy_url`)
: 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.

$ua->no_proxy(`$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', 'example.com');
```

$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="localhost,example.com"
 export gopher_proxy wais_proxy no_proxy
```

csh or tcsh users should use the `setenv` command to define these environment variables.

On systems with case insensitive environment variables there exists a name clash between the CGI environment variables and the `HTTP_PROXY` environment variable normally picked up by *env_proxy()*. Because of this `HTTP_PROXY` is not honored for CGI scripts. The `CGI_HTTP_PROXY` environment variable can be used instead.

Handlers

Handlers are code that injected at various phases during the processing of requests. The following methods are provided to manage the active handlers:

$ua->add_handler($phase => &cb, %matchspec)
Add handler to be invoked in the given processing phase. For how to specify `%matchspec` see "Matching" in HTTP::Config.

The possible values `$phase` and the corresponding callback signatures are:

request_preprepare => sub { my($request, `$ua`, `$h`) = `@_`; ... }
The handler is called before the `request_prepare` and other standard initialization of the request. This can be used to set up headers and attributes that the `request_prepare` handler depends on. Proxy initialization should take place here; but in general don't register handlers for this phase.

request_prepare => sub { my($request, `$ua`, `$h`) = `@_`; ... }
The handler is called before the request is sent and can modify the request any way it see fit. This can for instance be used to add certain headers to specific requests.

The method can assign a new request object to `$_`[0] to replace the request that is sent fully.

The return value from the callback is ignored. If an exception is raised it will abort the request and make the request method return a "400 Bad request" response.

request_send => sub { my($request, `$ua`, `$h`) = `@_`; ... }
This handler gets a chance of handling requests before they're sent to the protocol handlers. It should return an HTTP::Response object if it wishes to terminate the processing; otherwise it should return nothing.

The `response_header` and `response_data` handlers will not be invoked for this response, but the `response_done` will be.

response_header => sub { my($response, `$ua`, `$h`) = `@_`; ... }
This handler is called right after the response headers have been received, but before any content data. The handler might set up handlers for data and might croak to abort the request.

The handler might set the `$response`->{default_add_content} value to control if any received data should be added to the response object directly. This will initially be false if the `$ua`->*request()* method was called with a `$content_file` or `$content_cb` argument; otherwise true.

response_data => sub { my($response, `$ua`, `$h`, `$data`) = `@_`; ... }
This handler is called for each chunk of data received for the response. The handler might croak to abort the request.

This handler needs to return a TRUE value to be called again for subsequent chunks for the same request.

response_done => sub { my($response, `$ua`, `$h`) = `@_`; ... }
The handler is called after the response has been fully received, but before any redirect handling is attempted. The handler can be used to extract information or modify the response.

response_redirect => sub { my($response, `$ua`, `$h`) = `@_`; ... }
The handler is called in `$ua`->request after `response_done`. If the handler returns an HTTP::Request object we'll start over with processing this request instead.

$ua->remove_handler(undef, %matchspec)
$ua->remove_handler($phase, %matchspec)
Remove handlers that match the given `%matchspec`. If `$phase` is not provided remove handlers from all phases.

Be careful as calling this function with `%matchspec` that is not specific enough can remove handlers not owned by you. It's probably better to use the *set_my_handler()* method instead.

The removed handlers are returned.

$ua->set_my_handler(`$phase`, `$cb`, `%matchspec`)
: Set handlers private to the executing subroutine. Works by defaulting an `owner` field to the `%matchspec` that holds the name of the called subroutine. You might pass an explicit `owner` to override this.

  If `$cb` is passed as `undef`, remove the handler.

$ua->get_my_handler(`$phase`, `%matchspec`)
$ua->get_my_handler(`$phase`, `%matchspec`, `$init`)
: Will retrieve the matching handler as hash ref.

  If `$init` is passed as a TRUE value, create and add the handler if it's not found. If `$init` is a subroutine reference, then it's called with the created handler hash as argument. This sub might populate the hash with extra fields; especially the callback. If `$init` is a hash reference, merge the hashes.

$ua->handlers(`$phase`, `$request`)
$ua->handlers(`$phase`, `$response`)
: Returns the handlers that apply to the given request or response at the given processing phase.

REQUEST METHODS

The methods described in this section are used to dispatch requests via the user agent. The following request methods are provided:

$ua->get(`$url`)
$ua->get(`$url` , `$field_name` => `$value`, ...)
: This method will dispatch a `GET` request on the given `$url`. Further arguments can be given to initialize the headers of the request. These are given as separate name/value pairs. The return value is a response object. See HTTP::Response for a description of the interface it provides.

  There will still be a response object returned when LWP can't connect to the server specified in the URL or when other failures in protocol handlers occur. These internal responses use the standard HTTP status codes, so the responses can't be differentiated by testing the response status code alone. Error responses that LWP generates internally will have the "Client-Warning" header set to the value "Internal response". If you need to differentiate these internal responses from responses that a remote server actually generates, you need to test this header value.

  Fields names that start with ":" are special. These will not initialize headers of the request but will determine how the response content is treated. The following special field names are recognized:

```
:content_file   => $filename
:content_cb     => \&callback
:read_size_hint => $bytes
```

  If a `$filename` is provided with the `:content_file` option, then the response content will be saved here instead of in the response object. If a callback is provided with the `:content_cb` option then this function will be called for each chunk of the response content as it is received from the server. If neither of these options are given, then the response content will accumulate in the response object itself. This might not be suitable for very large response bodies. Only one of `:content_file` or `:content_cb` can be specified. The content of unsuccessful responses will always accumulate in the response object itself, regardless of the `:content_file` or `:content_cb` options passed in.

  The `:read_size_hint` option is passed to the protocol module which will try to read data from the server in chunks of this size. A smaller value for the `:read_size_hint` will result in a higher number of callback invocations.

The callback function is called with 3 arguments: a chunk of data, a reference to the response object, and a reference to the protocol object. The callback can abort the request by invoking *die()*. The exception message will show up as the ''X-Died'' header field in the response returned by the *get()* function.

$ua->head(`$url`)
$ua->head(`$url` , `$field_name` => `$value`, ...)
: This method will dispatch a `HEAD` request on the given `$url`. Otherwise it works like the *get()* method described above.

$ua->post(`$url`, %form)
$ua->post(`$url`, @form)
$ua->post(`$url`, %form, `$field_name` => `$value`, ...)
$ua->post(`$url`, `$field_name` => `$value`,... Content => %form)
$ua->post(`$url`, `$field_name` => `$value`,... Content => @form)
$ua->post(`$url`, `$field_name` => `$value`,... Content => `$content`)
: This method will dispatch a `POST` request on the given `$url`, with `%form` or `@form` providing the key/value pairs for the fill-in form content. Additional headers and content options are the same as for the *get()* method.

This method will use the *POST()* function from `HTTP::Request::Common` to build the request. See HTTP::Request::Common for a details on how to pass form content and other advanced features.

$ua->put(`$url`, %form)
$ua->put(`$url`, @form)
$ua->put(`$url`, %form, `$field_name` => `$value`, ...)
$ua->put(`$url`, `$field_name` => `$value`,... Content => %form)
$ua->put(`$url`, `$field_name` => `$value`,... Content => @form)
$ua->put(`$url`, `$field_name` => `$value`,... Content => `$content`)
: This method will dispatch a `PUT` request on the given `$url`, with `%form` or `@form` providing the key/value pairs for the fill-in form content. Additional headers and content options are the same as for the *get()* method.

This method will use the *PUT()* function from `HTTP::Request::Common` to build the request. See HTTP::Request::Common for a details on how to pass form content and other advanced features.

$ua->delete(`$url`)
$ua->delete(`$url`, `$field_name` => `$value`, ...)
: This method will dispatch a `DELETE` request on the given `$url`. Additional headers and content options are the same as for the *get()* method.

This method will use the *DELETE()* function from `HTTP::Request::Common` to build the request. See HTTP::Request::Common for a details on how to pass form content and other advanced features.

$ua->mirror(`$url`, `$filename`)
: This method will get the document identified by `$url` and store it in file called `$filename`. If the file already exists, then the request will contain an ''If-Modified-Since'' header matching the modification time of the file. If the document on the server has not changed since this time, then nothing happens. If the document has been updated, it will be downloaded again. The modification time of the file will be forced to match that of the server.

The return value is the response object.

$ua->request(`$request`)

$ua->request(`$request`, `$content_file`)
$ua->request(`$request`, `$content_cb`)
$ua->request(`$request`, `$content_cb`, `$read_size_hint`)

This method will dispatch the given `$request` object. Normally this will be an instance of the `HTTP::Request` class, but any object with a similar interface will do. The return value is a response object. See HTTP::Request and HTTP::Response for a description of the interface provided by these classes.

The *request()* method will process redirects and authentication responses transparently. This means that it may actually send several simple requests via the *simple_request()* method described below.

The request methods described above; *get()*, *head()*, *post()* and *mirror()*, will all dispatch the request they build via this method. They are convenience methods that simply hides the creation of the request object for you.

The `$content_file`, `$content_cb` and `$read_size_hint` all correspond to options described with the *get()* method above.

You are allowed to use a CODE reference as `content` in the request object passed in. The `content` function should return the content when called. The content can be returned in chunks. The content function will be invoked repeatedly until it return an empty string to signal that there is no more content.

$ua->simple_request(`$request`)
$ua->simple_request(`$request`, `$content_file`)
$ua->simple_request(`$request`, `$content_cb`)
$ua->simple_request(`$request`, `$content_cb`, `$read_size_hint`)

This method dispatches a single request and returns the response received. Arguments are the same as for *request()* described above.

The difference from *request()* is that *simple_request()* will not try to handle redirects or authentication responses. The *request()* method will in fact invoke this method for each simple request it sends.

$ua->is_online

Tries to determine if you have access to the Internet. Returns TRUE if the built-in heuristics determine that the user agent is able to access the Internet (over HTTP). See also LWP::Online.

$ua->is_protocol_supported(`$scheme`)

You can use this method to test whether this user agent object supports the specified `scheme`. (The `scheme` might be a string (like 'http' or 'ftp') or it might be an URI object reference.)

Whether a scheme is supported, is determined by the user agent's `protocols_allowed` or `protocols_forbidden` lists (if any), and by the capabilities of LWP. I.e., this will return TRUE only if LWP supports this protocol *and* it's permitted for this particular object.

Callback methods

The following methods will be invoked as requests are processed. These methods are documented here because subclasses of `LWP::UserAgent` might want to override their behaviour.

$ua->prepare_request(`$request`)

This method is invoked by *simple_request()*. Its task is to modify the given `$request` object by setting up various headers based on the attributes of the user agent. The return value should normally be the `$request` object passed in. If a different request object is returned it will be the one actually processed.

The headers affected by the base implementation are; ‘‘User-Agent’’, ‘‘From’’, ‘‘Range’’ and ‘‘Cookie’’.

$ua->redirect_ok(`$prospective_request`, `$response`)

This method is called by *request()* before it tries to follow a redirection to the request in `$response`. This should return a TRUE value if this redirection is permissible. The `$prospective_request` will be the request to be sent if this method returns TRUE.

The base implementation will return FALSE unless the method is in the object's `requests_redirectable` list, FALSE if the proposed redirection is to a "file://..." URL, and TRUE otherwise.

$ua->get_basic_credentials(`$realm`, `$uri`, `$isproxy`)

This is called by *request()* to retrieve credentials for documents protected by Basic or Digest Authentication. The arguments passed in is the `$realm` provided by the server, the `$uri` requested and a boolean flag to indicate if this is authentication against a proxy server.

The method should return a username and password. It should return an empty list to abort the authentication resolution attempt. Subclasses can override this method to prompt the user for the information. An example of this can be found in `lwp-request` program distributed with this library.

The base implementation simply checks a set of pre-stored member variables, set up with the *credentials()* method.

$ua->progress(`$status`, `$request_or_response`)

This is called frequently as the response is received regardless of how the content is processed. The method is called with `$status` "begin" at the start of processing the request and with `$state` "end" before the request method returns. In between these `$status` will be the fraction of the response currently received or the string "tick" if the fraction can't be calculated.

When `$status` is "begin" the second argument is the request object, otherwise it is the response object.

SEE ALSO

See LWP for a complete overview of libwww-perl5. See lwpcook and the scripts *lwp-request* and *lwp-download* for examples of usage.

See HTTP::Request and HTTP::Response for a description of the message objects dispatched and received. See HTTP::Request::Common and HTML::Form for other ways to build request objects.

See WWW::Mechanize and WWW::Search for examples of more specialized user agents based on `LWP::UserAgent`

COPYRIGHT