NAME

HTTP::Daemon − A simple http server class

VERSION

version 6.13

SYNOPSIS

```
use HTTP::Daemon;
use HTTP::Status;

my $d = HTTP::Daemon->new || die;
print "Please contact me at: <URL:", $d->url, ">\n";
while (my $c = $d->accept) {
    while (my $r = $c->get_request) {
    if ($r->method eq 'GET' and $r->uri->path eq "/xyzzy") {
            # remember, this is *not* recommended practice :-)
        $c->send_file_response("/etc/passwd");
    }
    else {
        $c->send_error(RC_FORBIDDEN)
    }
    }
    $c->close;
    undef($c);
}
```

DESCRIPTION

Instances of the `HTTP::Daemon` class are HTTP/1.1 servers that listen on a socket for incoming requests. The `HTTP::Daemon` is a subclass of `IO::Socket::IP`, so you can perform socket operations directly on it too.

Please note that `HTTP::Daemon` used to be a subclass of `IO::Socket::INET`. To support IPv6, it switched the parent class to `IO::Socket::IP` at version 6.05. See "IPv6 SUPPORT" for details.

The **accept()** method will return when a connection from a client is available. The returned value will be an `HTTP::Daemon::ClientConn` object which is another `IO::Socket::IP` subclass. Calling the **get_request()** method on this object will read data from the client and return an `HTTP::Request` object. The ClientConn object also provide methods to send back various responses.

This HTTP daemon does not **fork**(2) for you. Your application, i.e. the user of the `HTTP::Daemon` is responsible for forking if that is desirable. Also note that the user is responsible for generating responses that conform to the HTTP/1.1 protocol.

The following methods of `HTTP::Daemon` are new (or enhanced) relative to the `IO::Socket::IP` base class:

`$d` = HTTP::Daemon−>new
`$d` = HTTP::Daemon−>new(`%opts`)

The constructor method takes the same arguments as the `IO::Socket::IP` constructor, but unlike its base class it can also be called without any arguments. The daemon will then set up a listen queue of 5 connections and allocate some random port number.

A server that wants to bind to some specific address on the standard HTTP port will be constructed like this:

```
$d = HTTP::Daemon->new(
         LocalAddr => 'www.thisplace.com',
         LocalPort => 80,
     );
```

See IO::Socket::IP for a description of other arguments that can be used to configure the daemon

during construction.

$c = $d->accept
$c = $d->accept($pkg)
($c, $peer_addr) = $d->accept
This method works the same as the one provided by the base class, but it returns an HTTP::Daemon::ClientConn reference by default. If a package name is provided as argument, then the returned object will be blessed into the given class. It is probably a good idea to make that class a subclass of HTTP::Daemon::ClientConn.

The accept method will return undef if timeouts have been enabled and no connection is made within the given time. The **timeout()** method is described in IO::Socket::IP.

In list context both the client object and the peer address will be returned; see the description of the accept method of IO::Socket for details.

$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 main reason to have this method is that subclasses can override it if they want to use another product name.

The default is the string ''libwww-perl-daemon/#.##'' where ''#.##'' is replaced with the version number of this module.

The HTTP::Daemon::ClientConn is a subclass of IO::Socket::IP. Instances of this class are returned by the **accept()** method of HTTP::Daemon. The following methods are provided:

$c->get_request
$c->get_request($headers_only)
This method reads data from the client and turns it into an HTTP::Request object which is returned. It returnsundef if reading f ails. If it fails, then the HTTP::Daemon::ClientConn object ($c) should be discarded, and you should not try to call this method again on it. The $c->reason method might give you some information about why $c->get_request failed.

The **get_request()** method will normally not return until the whole request has been received from the client. This might not be what you want if the request is an upload of a large file (and with chunked transfer encoding HTTP can even support infinite request messages - uploading live audio for instance). If you pass a TRUE value as the $headers_only argument, then **get_request()** will return immediately after parsing the request headers and you are responsible for reading the rest of the request content. If you are going to call $c->get_request again on the same connection you better read the correct number of bytes.

$c->read_buffer
$c->read_buffer($new_value)
Bytes read by $c->get_request, but not used are placed in the *read buffer*. The next time $c->get_request is called it will consume the bytes in this buffer before reading more data from the network connection itself. The read buffer is invalid after $c->get_request has failed.

If you handle the reading of the request content yourself you need to empty this buffer before you read more and you need to place unconsumed bytes here. You also need this buffer if you implement services like *101 Switching Protocols*.

This method always returns the old buffer content and can optionally replace the buffer content if you pass it an argument.

$c->reason
When $c->get_request returns undef you can obtain a short string describing why it happened by calling $c->reason.

$c->proto_ge($proto)
: Return TRUE if the client announced a protocol with version number greater or equal to the given argument. The$proto ar gument can be a string like ''HTTP/1.1'' or just ''1.1''.

$c->antique_client
: Return TRUE if the client speaks the HTTP/0.9 protocol. No status code and no headers should be returned to such a client. This should be the same as !$c->proto_ge(''HTTP/1.0'').

$c->head_request
: Return TRUE if the last request was a HEAD request. No content body must be generated for these requests.

$c->force_last_request
: Make sure that $c->get_request will not try to read more requests off this connection. If you generate a response that is not self-delimiting, then you should signal this fact by calling this method.

  This attribute is turned on automatically if the client announces protocol HTTP/1.0 or worse and does not include a ''Connection: Keep-Alive'' header. It is also turned on automatically when HTTP/1.1 or better clients send the ''Connection: close'' request header.

$c->send_status_line
$c->send_status_line($code)
$c->send_status_line($code, $mess)
$c->send_status_line($code, $mess, $proto)
: Send the status line back to the client. If $code is omitted 200 is assumed. If $mess is omitted, then a message corresponding to $code is inserted. If $proto is missing the content of the $HTTP::Daemon::PROTO variable is used.

$c->send_crlf
: Send the CRLF sequence to the client.

$c->send_basic_header
$c->send_basic_header($code)
$c->send_basic_header($code, $mess)
$c->send_basic_header($code, $mess, $proto)
: Send the status line and the ''Date:'' and ''Server:'' headers back to the client. This header is assumed to be continued and does not end with an empty CRLF line.

  See the description of **send_status_line()** for the description of the accepted arguments.

$c->send_header($field, $value)
$c->send_header($field1, $value1, $field2, $value2, ...)
: Send one or more header lines.

$c->send_response($res)
: Write an HTTP::Response object to the client as a response. We try hard to make sure that the response is self-delimiting so that the connection can stay persistent for further request/response exchanges.

  The content attribute of the HTTP::Response object can be a normal string or a subroutine reference. If it is a subroutine, then whatever this callback routine returns is written back to the client as the response content. The routine will be called until it returns an undefined or empty value. If the client is HTTP/1.1 aware then we will use chunked transfer encoding for the response.

$c->send_redirect($loc)
$c->send_redirect($loc, $code)
$c->send_redirect($loc, $code, $entity_body)
: Send a redirect response back to the client. The location ($loc) can be an absolute or relative URL. The $code must be one of the redirect status codes, and defaults to ''301 Moved Permanently''

`$c->send_error`
`$c->send_error( $code )`
`$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.

`$c->send_file_response( $filename )`
Send back a response with the specified `$filename` as content. If the file is a directory we try to generate an HTML index of it.

`$c->send_file( $filename )`
`$c->send_file( $fd )`
Copy the file to the client. The file can be a string (which will be interpreted as a filename) or a reference to an `IO::Handle` or glob.

`$c->daemon`
Return a reference to the corresponding `HTTP::Daemon` object.

IPv6 SUPPORT

Since version 6.05, `HTTP::Daemon` is a subclass of `IO::Socket::IP` rather than `IO::Socket::INET`, so that it supports IPv6.

For some reasons, you may want to force `HTTP::Daemon` to listen on IPv4 addresses only. Then pass `Family` argument to `HTTP::Daemon->new`:

```
use HTTP::Daemon;
use Socket 'AF_INET';

my $d = HTTP::Daemon->new(Family => AF_INET);
```

SEE ALSO

RFC 2616

IO::Socket::IP, IO::Socket

SUPPORT

Bugs may be submitted through <https://github.com/libwww-perl/HTTP-Daemon/issues>.

There is also a mailing list available for users of this distribution, at <mailto:libwww@perl.org>.

There is also an irc channel available for users of this distribution, at `#lwp` on `irc.perl.org` <irc://irc.perl.org/#lwp>.

AUTHOR

Gisle Aas <gisle@activestate.com>

CONTRIBUTORS

- Olaf Alders <olaf@wundersolutions.com>
- Ville Skyttä <ville.skytta@iki.fi>
- Karen Etheridge <ether@cpan.org>
- Mark Stosberg <MARKSTOS@cpan.org>
- Shoichi Kaji <skaji@cpan.org>
- Chase Whitener <capoeirab@cpan.org>
- Slaven Rezic <slaven@rezic.de>
- Petr PísaX <ppisar@redhat.com>
- Zefram <zefram@fysh.org>
- Alexey Tourbin <at@altlinux.ru>

- Bron Gondwana <brong@fastmail.fm>
- Mike Schilli <mschilli@yahoo−inc.com>
- Tom Hukins <tom@eborcom.com>
- Adam Kennedy <adamk@cpan.org>
- Adam Sjogren <asjo@koldfront.dk>
- Alex Kapranoff <ka@nadoby.ru>
- amire80 <amir.aharoni@gmail.com>
- Andreas J. Koenig <andreas.koenig@anima.de>
- Bill Mann <wfmann@alum.mit.edu>
- Daniel Hedlund <Daniel.Hedlund@eprize.com>
- David E. Wheeler <david@justatheory.com>
- DAVIDRW <davidrw@cpan.org>
- Father Chrysostomos <sprout@cpan.org>
- Ferenc Erki <erkiferenc@gmail.com>
- FWILES <FWILES@cpan.org>
- Gavin Peters <gpeters@deepsky.com>
- Graeme Thompson <Graeme.Thompson@mobilecohesion.com>
- Hans-H. Froehlich <hfroehlich@co−de−co.de>
- Ian Kilgore <iank@cpan.org>
- Jacob J <waif@chaos2.org>
- jefflee <shaohua@gmail.com>
- john9art <john9art@yahoo.com>
- murphy <murphy@genome.chop.edu>
- Ondrej Hanak <ondrej.hanak@ubs.com>
- Perlover <perlover@perlover.com>
- Peter Rabbitson <ribasushi@cpan.org>
- phrstbrn <phrstbrn@gmail.com>
- Robert Stone <talby@trap.mtview.ca.us>
- Rolf Grossmann <rg@progtech.net>
- ruff <ruff@ukrpost.net>
- sasao <sasao@yugen.org>
- Sean M. Burke <sburke@cpan.org>
- Spiros Denaxas <s.denaxas@gmail.com>
- Steve Hay <SteveHay@planit.com>
- Todd Lipcon <todd@amiestreet.com>
- Tony Finch <dot@dotat.at>
- Toru Yamaguchi <zigorou@cpan.org>
- Yuri Karaban <tech@askold.net>

COPYRIGHT AND LICENCE