Inets

version 4.5
## Contents

### 1 Inets User’s Guide

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>1.1.1</td>
<td>Purpose</td>
</tr>
<tr>
<td>1.1.2</td>
<td>Prerequisites</td>
</tr>
<tr>
<td>1.1.3</td>
<td>The Service Concept</td>
</tr>
<tr>
<td>1.2</td>
<td>FTP client</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Using the ftp client API</td>
</tr>
<tr>
<td>1.3</td>
<td>HTTP client</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Configuration</td>
</tr>
<tr>
<td>1.3.3</td>
<td>Using the HTTP client API</td>
</tr>
<tr>
<td>1.4</td>
<td>HTTP server</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>1.4.2</td>
<td>HTTP Server Setup</td>
</tr>
<tr>
<td>1.4.3</td>
<td>Run-Time Configuration</td>
</tr>
<tr>
<td>1.4.4</td>
<td>Erlang Web Server API</td>
</tr>
</tbody>
</table>
Chapter 1

Inets User's Guide

The Inets Application provides a set of Internet related services. Currently supported are a HTTP client, a HTTP server and a ftp client.

1.1 Introduction

1.1.1 Purpose

Inets is a container for Internet clients and servers. Currently, an HTTP server, a FTP client and an HTTP client has been incorporated into Inets. The HTTP server and client is HTTP 1.1 compliant as defined in RFC 2616.

The HTTP client is now documented http(3) [page 35] and supported. It is loosely based on the earlier unsupported client developed by Johan Blom of Mobile Arts AB.

1.1.2 Prerequisites

It is assumed that the reader is familiar with the Erlang programming language, concepts of OTP and has a basic understanding of the HTTP and FTP protocols.

1.1.3 The Service Concept

Each client and server in inets is viewed as service. When starting the inets application the inets top supervisor will start a number of subsupervisors and worker processes for handling the different services provided. Some services require that there exist a configuration file, such as HTTP server(s), in order for the service(s) to be started. While the HTTP clients main process always will be started (it remains idle until some process issues a request) in this case the configuration is optional. Other services may not be configurable and have a more dynamic character, such as ftp clients, that will add worker processes to the supervision tree every time you do ftp:open/[1,2,3] an remove them every time you do ftp:close/1.

Services that needs configuring should be put into the inets applications configuration file on the form:

    [{inets, [{services, ListofConfiguredServices}]}].

For details of exactly what to put in the list of configured services see the documentation for the services that needs configuring.
Chapter 1: Inets User's Guide

1.2 FTP client

1.2.1 Introduction

Ftp client processes exist only when you use them. When you open a ftp connection a client process will be spawned and added as a dynamic child to the ftp supervisor in the inets supervision tree. When you close the connection the client process will be terminated. Only the process that created the ftp-connection will be permitted to use it, and if that process dies the connection process will terminate. The client supports ipv6 as long as the underlying mechanisms also do so.

1.2.2 Using the ftp client API

The following is a simple example of an ftp session, where the user guest with password password logs on to the remote host erlang.org, and where the file appl.erl is transferred from the remote to the local host. When the session is opened, the current directory at the remote host is /home/guest, and /home/fred at the local host. Before transferring the file, the current local directory is changed to /home/eproj/examples, and the remote directory is set to /home/guest/appl/examples.

```
1> application:start(inets).
ok
2> {ok, Pid} = ftp:open("erlang.org").
{ok,<0.22.0>}
3> ftp:user(Pid, "guest", "password").
ok
4> ftp:pwd(Pid).
{ok, "/home/guest"}
5> ftp:cd(Pid, "appl/examples").
ok
6> ftp:lpwd(Pid).
{ok, "/home/fred"}.
7> ftp:lcd(Pid, "/home/eproj/examples").
ok
8> ftp:recv(Pid, "appl.erl").
ok
9> ftp:close(Pid).
ok
```

1.3 HTTP client

1.3.1 Introduction

The HTTP client will be started when the inets application is started and is then available to all processes on that erlang node. The client will spawn a new process to handle each request unless there is a possibility to pipeline a request. The client will add a host header and an empty te header if there are no such headers present in the request. The client supports ipv6 as long as the underlying mechanisms also do so.
1.3.2 Configuration

It is possible to configure what directory the HTTP client should use to store information. Currently the only information stored here is cookies. If the HTTP client service is not configured all cookies will be treated as session cookies. Here follows a description of a configuration entry for the HTTP client in the application configuration file.

```
[{inets, [{services, [{httpc, {Profile, Dir}}]}]}]
```

Profile = atom() - default is the only valid value, as profiles are currently not supported.
Dir = string()

1.3.3 Using the HTTP client API

1 > application:start(inets).
ok

Use the proxy “www-proxy.mycompany.com:8000”, but not for requests to localhost. This will apply to all subsequent requests

2 > http:set_options([{{proxy, {"www-proxy.mycompany.com", 8000}, ["localhost"]}}]).
ok

An ordinary synchronous request.

2 > {ok, {{Version, 200, ReasonPhrase}, Headers, Body}} =
   http:request(get, {"http://www.erlang.org", [], [], []}).

An ordinary asynchronous request. The result will be sent to the calling process on the form {http, {RequestId, Result}}

3 > {ok, RequestId} =
   http:request(get, {"http://www.erlang.org", [], [], [{sync, false}]}).

In this case the calling process is the shell, so we receive the result.

4 > receive {http, {RequestId, Result}} -> ok after 500 -> error end.
ok

Send a request with a specified connection header.

5 > {ok, {{NewVersion, 200, NewReasonPhrase}, NewHeaders, NewBody}} =
   http:request(get, {"http://www.erlang.org", [{"connection", "close"}]}),
   [], []).
Chapter 1: Inets User's Guide

1.4 HTTP server

1.4.1 Introduction

The HTTP server also referred to as httpd handles HTTP requests as described in RFC 2616 with a few exceptions such as Gateway and Proxy functionality. (The same is true for servers written by NCSA and others.) The server supports IPv6 as long as the underlying mechanisms also do so.

The server implements numerous features such as SSL [page 48] (Secure Sockets Layer), ESI [page 88] (Erlang Scripting Interface), CGI [page 80] (Common Gateway Interface), User Authentication [page 68] (using Mnesia, dets or plain text database), Common Logfile Format (with [page 84] or without [page 104] disk_{log}(3) support), URL Aliasing [page 65], Action Mappings [page 63], Directory Listings [page 83] and SSI [page 101] (Server-Side Includes).

The configuration of the server is done using Apache1-style configuration directives. The goal is to be plug-in compatible with Apache.

Almost all server functionality has been implemented using an especially crafted server API, it is described in the Erlang Web Server API. This API can be used to advantage by all who wants to enhance the server core functionality, for example custom logging and authentication.

1.4.2 HTTP Server Setup

It is possible to start a number of Web servers in an embedded system using the services config parameter from an application config file. A minimal application config file (from now on referred to as inets.config) starting two HTTP servers typically looks as follows:

```erlang
[[inets,
  [services,[[httpd,"/var/tmp/server_root/conf/8888.conf"],
  [httpd,"/var/tmp/server_root/conf/8080.conf"]]]]].
```

A server config file is specified for each HTTP server to be started. The config file syntax and semantics is described in the run time configuration section.

inets.config can be tested by copying the example server root to a specific installation directory, as described in the run time configuration section. The example below shows a manual start of an Erlang node, using inets.config, and the start of two HTTP servers listening on ports 8888 and 8080.

```
$ erl -config ./inets
  Erlang (BEAM) emulator version 4.9

  Eshell V4.9  (abort with ^G)
  1> application:start(inets).
  ok
```

1URL: http://www.apache.org
1.4.3 Run-Time Configuration

All functionality in the server can be configured using Apache-style configuration directives stored in a configuration file. Take a look at the example config files in the conf directory2 of the server root for a complete understanding.

An alphabetical list of all config directives:

- AccessFileName [page 96]
- Action [page 63]
- Alias [page 65]
- allow [page 72]
- AuthName [page 71]
- AuthGroupFile [page 71]
- AuthUserFile [page 70]
- BindAddress [page 48]
- DefaultType [page 49]
- deny [page 72]
- Directory [page 68]
- DirectoryIndex [page 65]
- DisableChunkedTransferEncodingSend [page 49]
- DocumentRoot [page 49]
- ErlScriptAlias [page 91]
- ErlScriptNoCache [page 91]
- ErlScriptTimeout [page 91]
- ErrorLog [page 104]
- ErrorDiskLog [page 85]
- ErrorDiskLogSize [page 85]
- EvalScriptAlias [page 92]
- KeepAlive [page 49]
- KeepAliveTimeout [page 50]
- MaxBodySize [page 50]
- MaxBodyAction [page 50]
- MaxClients [page 50]
- MaxHeaderValue [page 51]
- MaxHeaderAction [page 51]
- MaxKeepAliveRequests [page 51]
- Modules [page 51]
- Port [page 52]
- require [page 73]
- SecurityAuthTimeout [page 111]
- SecurityBlockTime [page 110]

2In Windows: %INETS_ROOT%\examples\server_root\conf\ In UNIX: $INETS_ROOT/examples/server_root/conf/.
1.4.4 Erlang Web Server API

Almost all server functionality has been implemented using EWSAPI (Erlang Web Server API) modules. The following modules are available:

httpd_core [page 47] Some core features are not implemented using EWSAPI you can read about these int the virtual module httpd_core.
mod_alias [page 65] Aliases and redirects.
mod_auth [page 68] User authentication using text files, mnesia or dets.
mod_browser [page 79] Tries to recognize the clients browser and operating system.
mod_cgi [page 80] Invoking of CGI scripts.
mod_dir [page 83] Basic directory handling.
mod_disk_log [page 84] Standard logging in the Common Logfile Format using disk_log(3).
1.4: HTTP server

mod_get [page 94] Handle HTTP GET Method.
mod_head [page 95] Handle HTTP HEAD Method.
mod_htacceess [page 96] User configurable user authentication.
mod_include [page 101] Server-parsed documents.
mod_range [page 107] Handles GET requests for parts of files.
mod_responsecontrol [page 108] Controls the restrictions in the request i.e. If-Match, If-Range, If-Modified-Since, and take the appropriate action.
mod_trace [page 114] Handles HTTP TRACE Method.

Each module has a man page that further describe it’s functionality.
The Modules [page 51] config directive can be used to alter the server behavior, by alter the EWSAPI Module Sequence. An example module sequence can be found in the example config directory. If this needs to be altered read the EWSAPI Module Interaction section below.

EWSAPI Module Programming

Note:
The Erlang/OTP programming knowledge required to undertake an EWSAPI module is quite high and is not recommended for the average server user. It is best to only use it to add core functionality, e.g. custom authentication or a RFC 2109\(^3\) implementation.

EWSAPI should only be used to add core functionality to the server. In order to generate dynamic content, for example on-the-fly generated HTML, use the standard CGI [page 80] or ESI [page 88] facilities instead.

As seen above the major part of the server functionality has been realized as EWSAPI modules (from now on only called modules). If you intend to write your own server extension start with examining the standard modules\(^4\) mod_*\(\).erl and note how to they are configured in the example config directory\(^5\).

Each module implements do/1 (mandatory), load/2, store/2 and remove/1. The latter functions are needed only when new config directives are to be introduced.

A module can choose to export functions to be used by other modules in the EWSAPI Module Sequence (See Modules [page 51] config directive). This should only be done as an exception! The goal is to keep each module self-sustained thus making it easy to alter the EWSAPI Module Sequence without any unnecessary module dependencies.

A module can furthermore use data generated by previous modules in the EWSAPI Module Sequence or generate data to be used by consecutive EWSAPI modules. This is made possible due to an internal list of key-value tuples.

\(^4\)In Windows: %INETS\ROOT\src\ In UNIX: $INETS\ROOT/src/.
\(^5\)In Windows: %INETS\ROOT\examples\server_root\conf\ In UNIX: $INETS\ROOT/examples/server_root/conf/.

Inets  7
Note:
The server executes do/1 function of each module listed in the Modules [page 51] config directive. do/1 takes the record mod as an argument, as described below. See httpd.hrl:

```
-record(mod, {data=[],
        socket_type=ip_comm,
        socket, 
        config_db, 
        method, 
        absolute_uri,
        request_uri, 
        http_version, 
        request_line, 
        parsed_header=[],
        entity_body, 
        connection}).
```

The fields of the mod record has the following meaning:

data Type [[InteractionKey,InteractionValue]] is used to propagate data between modules. Depicted interaction_data() in function type declarations.

socket_type socket_type(), Indicates whether it is a ip socket or a ssl socket.

socket The actual socket in ip_comm or ssl format depending on the socket_type.

config_db The config file directives stored as key-value tuples in an ETS-table. Depicted config_db() in function type declarations.

method Type "GET" | "POST" | "HEAD" | "TRACE", that is the HTTP method.

absolute_uri If the request is a HTTP/1.1 request the URI might be in the absolute URI format. In that case httpd will save the absolute URI in this field. An Example of an absolute URI could be "http://ServerName:Part/cgi-bin/find.pl?person=jocke"

request_uri The Request-URI as defined in RFC 1945, for example "/cgi-bin/find.pl?person=jocke"

http_version The HTTP version of the request, that is "HTTP/0.9", "HTTP/1.0", or "HTTP/1.1".

request_line The Request-Line as defined in RFC 1945, for example "GET
"/cgi-bin/find.pl?person=jocke HTTP/1.0".

parsed_header Type [[HeaderKey,HeaderValue]], parsed_header contains all HTTP header fields from the HTTP-request stored in a list as key-value tuples. See RFC 2616 for a listing of all header fields. For example the date field would be stored as: ["date","Wed, 15 Oct 1997 14:35:17 GMT"]. RFC 2616 defines that HTTP is a case insensitive protocol and the header fields may be in lowercase or upper case. Httpd will ensure that all header field names are in lowercase.

entity_body The Entity-Body as defined in RFC 2616, for example data sent from a CGI-script using the POST method.

connection true | false If set to true the connection to the client is a persistent connections and will not be closed when the request is served.
A do/1 function typically uses a restricted set of the mod record's fields to do its stuff and then returns a term depending on the outcome. The outcome is either \{proceed,NewData\} | \{break,NewData\} | done. Which has the following meaning:

\{proceed,OldData\}  Proceed to next module as nothing happened. \(\text{OldData}\) refers to the data field in the incoming mod record.

\{proceed,[[response,{StatusCode,Response}]]\|OldData\}  A generated response (Response) should be sent back to the client including a status code (StatusCode) as defined in RFC 2616.

\{proceed,[[response,{{response,Head,Body}}]|OldData\}}  Head is a list of key/value tuples. Each HTTP-header field that will be in the response header must be in the list. The following atoms are allowed header field keys:

- code,
- allow,
- cache_control,
- content_MD5,
- content_encoding,
- content_language,
- content_length,
- content_location,
- content_range,
- content_type,
- date,
- etag,
- expires,
- last_modified
- location,
- pragma,
- retry_after,
- server,
- trailer,
- transfer_encoding,

The key code is a special case since the value to this key is an integer and not a string. The value will be used as status code for the response.

The benefit of this method is that the same request may be generated for both HTTP/1.1 and HTTP/1.0 clients since the list of header fields will be filtered due to the version of the request. Body is either the tuple \{Fun,Arg\} a list or the atom nobody. If Body is \{Fun,Arg\} Fun is assumed to be a fun that returns either close, sent or \{ok,Body\}. If close is returned the connection to the client will be closed. If sent is returned the connection to the client will be maintained if the connection is persistent. If \{ok,Body\} is returned the Body is sent back to the client as the response body.

This is the preferred response since it makes it a lot easier to generate a response that can be sent back to both HTTP/1.0 and HTTP/1.1 clients. A warning might be in place that if content_length is not sent to the client, it might hang unless the body is sent with chunked encoding.

\{proceed,[[response,{{already_sent,StatusCode,Size}}]|OldData\}}  A generated response has already manually been sent back to the client, using the socket provided by the mod record (see above), including a valid status code (StatusCode) as defined in RFC 1945 and the size (Size) of the response in bytes.

\{proceed,[[status,{{StatusCode,PhraseArgs,Reason}}]|OldData\}}  A generic status message
should be sent back to the client (if the next module in the EWS API Module Sequence does not think otherwise!) including at status code (StatusCode) as defined in RFC 1945, a term describing how the client will be informed (PhraseArgs) and a reason (Reason) to why it happened. Read more about PhraseArgs in httpd_util:message/3 [page 60].

{break, NewData} Has the same semantics as proceed above but with one important exception; No more modules in the EWS API Module Sequence are executed. Use with care!
done No more modules in the EWS API Module Sequence are executed and no response should be sent back to the client. If no response is sent back to the client, using the socket provided by the mod record, the client will typically get a “Document contains no data...”.

Warning:
Each consecutive module in the EWS API Module Sequence can choose to ignore data returned from the previous module either by trashing it or by “enhancing” it.

Keep in mind that there exist numerous utility functions to help you as an EWS API module programmer, e.g. nifty lookup of data in ETS-tables/key-value lists and socket utilities. You are well advised to read httpd_util(3) [page 57] and httpd_socket(3) [page 56].

EWSAPI Module Configuration

An EWSAPI module can define new config directives thus making it configurable for a server end-user. This is done by implementing load/2 (mandatory), store/2 and remove/1.

The config file is scanned twice (load/2 and store/2) and a cleanup is done (remove/1) during server shutdown. The reason for this is: “A directive A can be dependent upon another directive B which occur either before or after directive A in the config file”. If a directive does not depend upon other directives; store/2 can be left out. Even remove/1 can be left out if neither load/2 nor store/2 open files or create ETS-tables etc.

load/2 takes two arguments. The first being a row from the config file, that is a config directive in string format such as "Port 80". The second being a list of key-value tuples (which can be empty!) defining a context. A context is needed because there are directives which defines inner contexts, that is directives within directives, such as <Directory> [page 68]. load/2 is expected to return:

eof End-of-file found.
ok Ignore the directive.
{ok, ContextList} Introduces a new context by adding a tuple to the context list or reverts to a previous context by removing a tuple from the context list. See <Directory> [page 68] which introduces a new context and </Directory> [page 68] which reverts to a previous one (Advice: Look at the source code for mod_auth:load/2).
{ok, ContextList, [DirectiveKey, DirectiveValue]} Introduces a new context (see above) and defines a new config directive, e.g. {port, 80}.
{ok, ContextList, [[DirectiveKey, DirectiveValue]]} Introduces a new context (see above) and defines a several new config directives, e.g. [[port, 80], [foo, on]].
{error, Reason} An invalid directive.

An example of a load function from mod_log.erl:
1.4: HTTP server

load("TransferLog " ++ TransferLog, []) ->
  {ok, [], {transfer_log, httpd_conf:clean(TransferLog)});

load("ErrorLog " ++ ErrorLog, []) ->
  {ok, [], {error_log, httpd_conf:clean(ErrorLog)}).

store/2 takes two arguments. The first being a tuple describing a directive
{{DirectiveKey,DirectiveValue}} and the second argument a list of tuples describing all directives
{{{DirectiveKey,DirectiveValue}}}]. This makes it possible for directive A to be dependent upon
the value of directive B. store/2 is expected to return:

{ok, {DirectiveKey, NewDirectiveValue}}] Introduces a new value for the specified directive
  replacing the old one generated by load/2.
{ok, [{DirectiveKey, NewDirectiveValue}]} Introduces new values for the specified directives
  replacing the old ones generated by load/2.
{error, Reason} A invalid directive.

An example of a store function from mod_log.erl:

store({error_log, ErrorLog}, ConfigList) ->
case create_log(ErrorLog, ConfigList) of
  {ok, ErrorLogStream} ->
    {ok, {error_log, ErrorLogStream}};
  {error, Reason} -> {error, Reason}
end.

remove/1 takes the ETS-table representation of the config-file as input. It is up to you to cleanup
anything you opened or created in load/2 or store/2. remove/1 is expected to return:

ok If the cleanup was successful.
{error, Reason} If the cleanup failed.

A naive example from mod_log.erl:

remove(ConfigDB) ->
  lists:foreach(fun([Stream]) -> file:close(Stream) end,
               ets:match(ConfigDB, {transfer_log, '$1'})),
  lists:foreach(fun([Stream]) -> file:close(Stream) end,
               ets:match(ConfigDB, {error_log, '$1'})),
  ok.
Chapter 1: Inets User's Guide

EWSAPI Module Interaction

Modules in the EWSAPI Module Sequence [page 51] use the mod record's data field to propagate responses and status messages, as seen above. This data type can be used in a more versatile fashion. A module can prepare data to be used by subsequent EWSAPI modules, for example the mod_alias [page 65] module appends the tuple \{real_name,string\} to inform subsequent modules about the actual file system location for the current URL.

Before altering the EWSAPI Modules Sequence you are well advised to observe what types of data each module uses and propagates. Read the “EWSAPI Interaction” section for each module.

An EWSAPI module can furthermore export functions to be used by other EWSAPI modules but also for other purposes, for example mod_alias:path/3 [page 67] and mod_auth:add_user/5 [page 74]. These functions should be described in the module documentation.

**Note:**

When designing an EWSAPI module try to make it self-contained, that is avoid being dependent on other modules both concerning exchange of interaction data and the use of exported functions. If you are dependent on other modules do state this clearly in the module documentation!

You are well advised to read httpd_util(3) [page 57] and httpd_conf(3) [page 45].
Inets Reference Manual

Short Summaries

- Erlang Module **ftp** [page 25] - A File Transfer Protocol client
- Erlang Module **http** [page 35] - A HTTP/1.1 client
- Erlang Module **http_base64** [page 39] - Implements base 64 encode and decode, see RFC 2045.
- Erlang Module **httpd** [page 40] - An implementation of an HTTP 1.1 compliant Web server, as defined in RFC 2616.
- Erlang Module **httpd_conf** [page 45] - Configuration utility functions to be used by the EWSAPI programmer.
- Erlang Module **httpd_core** [page 47] - The core functionality of the Web server.
- Erlang Module **httpd_socket** [page 56] - Communication utility functions to be used by the EWSAPI programmer.
- Erlang Module **httpd_util** [page 57] - Miscellaneous utility functions to be used when implementing EWSAPI modules.
- Erlang Module **mod_actions** [page 63] - Filetype/method-based script execution.
- Erlang Module **mod_alias** [page 65] - This module creates aliases and redirections
- Erlang Module **mod_auth** [page 68] - User authentication using text files, dets or mnesia database.
- Erlang Module **mod_browser** [page 79] - Tries to recognize the browser and operating-system of the client.
- Erlang Module **mod_cgi** [page 80] - Invoking of CGI scripts
- Erlang Module **mod_dir** [page 83] - Basic directory handling.
- Erlang Module **mod_disk_log** [page 84] - Standard logging using the "Common Logfile Format" and diskLog(3).
- Erlang Module **mod_esi** [page 88] - Efficient Erlang Scripting
- Erlang Module **mod_get** [page 94] - Handle GET requests.
- Erlang Module **mod_head** [page 95] - Handles HEAD requests to regular files.
- Erlang Module **mod_htaccess** [page 96] - This module provides per-directory user-configurable access control.
- Erlang Module **mod_include** [page 101] - Server-parsed documents.
- Erlang Module **mod_log** [page 104] - Standard logging using the "Common Logfile Format" and text files.
- Erlang Module **mod_range** [page 107] - handle requests for parts of a file
Inets Reference Manual

- **Erlang Module mod_responsecontrol** [page 108] - Controls that the request conditions is fullfilled.
- **Erlang Module mod_security** [page 109] - Security Audit and Trailing Functionality
- **Erlang Module mod_trace** [page 114] - handle trace requests

**ftp**

The following functions are exported:

- `account(Pid, Account) -> ok | {error, Reason}`
  [page 26] Specify which account to use.
- `append(Pid, LocalFile [, RemoteFile]) -> ok | {error, Reason}`
  [page 26] Transfer file to remote server, and append it to Remotefile.
- `append_bin(Pid, Bin, RemoteFile) -> ok | {error, Reason}`
  [page 26] Transfer a binary into a remote file.
- `append_chunk(Pid, Bin) -> ok | {error, Reason}`
  [page 26] append a chunk to the remote file.
- `append_chunk_start(Pid, File) -> ok | {error, Reason}`
  [page 26] Start transfer of file chunks for appending to File.
- `append_chunk_end(Pid) -> ok | {error, Reason}`
  [page 27] Stop transfer of chunks for appending.
- `cd(Pid, Dir) -> ok | {error, Reason}`
  [page 27] Change remote working directory.
- `close(Pid) -> ok` [page 27] End ftp session.
- `delete(Pid, File) -> ok | {error, Reason}`
  [page 27] Delete a file at the remote server.
- `formaterror(Tag) -> string()` [page 27] Return error diagnostics.
- `lpwd(Pid) -> {ok, Dir}` [page 27] Get remote current working directory.
- `lc(Pid, Dir) -> ok | {error, Reason}`
  [page 27] Get local current working directory.
- `ls(Pid [, Dir]) -> {ok, Listing} | {error, Reason}`
  [page 28] List contents of remote directory.
- `mkdir(Pid, Dir) -> ok | {error, Reason}`
  [page 28] Create remote directory.
- `nlist(Pid [, Dir]) -> {ok, Listing} | {error, Reason}`
  [page 28] List contents of remote directory.
- `open(Host [, Port] [, Flags]) -> {ok, Pid} | {error, Reason}`
- `open({option_list, Option_list}) -> {ok, Pid} | {error, Reason}`
- `pwd(Pid) -> {ok, Dir} | {error, Reason}`
- `recv(Pid, RemoteFile [, LocalFile])` -> `ok | {error, Reason}`  

- `recv_bin(Pid, RemoteFile) -> {ok, Bin} | {error, Reason}`  
  [page 30] Transfer file from remote server as a binary.

- `recv_chunk_start(Pid, RemoteFile) -> ok | {error, Reason}`  
  [page 31] Start chunk-reading of the remote file.

- `recv_chunk(Pid) -> ok | {ok, Bin} | {error, Reason}`  
  [page 31] Receive a chunk of the remote file.

- `rename(Pid, Old, New) -> ok | {error, Reason}`  
  [page 31] Rename a file at the remote server.

- `rmdir(Pid, Dir) -> ok | {error, Reason}`  
  [page 31] Remove a remote directory.

- `send(Pid, LocalFile [, RemoteFile])` -> `ok | {error, Reason}`  
  [page 31] Transfer file to remote server.

- `send_bin(Pid, Bin, RemoteFile) -> ok | {error, Reason}`  
  [page 32] Transfer a binary into a remote file.

- `send_chunk(Pid, Bin) -> ok | {error, Reason}`  
  [page 32] Write a chunk to the remote file.

- `send_chunk_start(Pid, File) -> ok | {error, Reason}`  

- `send_chunk_end(Pid) -> ok | {error, Reason}`  
  [page 32] Stop transfer of chunks.

- `type(Pid, Type) -> ok | {error, Reason}`  
  [page 32] Set transfer type to `ascii` or `binary`.

- `user(Pid, User, Password) -> ok | {error, Reason}`  

- `user(Pid, User, Password,Account) -> ok | {error, Reason}`  
  [page 33] User login.

- `quote(Pid, Command) -> [FTPLine]`  
  [page 33] Sends an arbitrary FTP command.

**http**

The following functions are exported:

- `cancel_request(RequestId) -> ok`  
  [page 36] Cancels an asynchronous HTTP-request.

- `request(Url) -> {ok, Result} | {error, Reason}`  
  [page 36] Sends a get HTTP-request.

- `request(Method, Request, HTTPOptions, Options) -> {ok, Result} | {error, Reason}`  
  [page 36] Sends a HTTP-request.

- `set_options(Options) -> ok`  
  [page 37] Sets options to be used for subsequent requests.

- `verify_cookie(SetCookieHeaders, Url) -> ok`  
  [page 38] Saves the cookies defined in `SetCookieHeaders` in the client managers cookie database.
- `cookie_header(Url)` -> `header()`  
  [page 38] Returns the cookie header that would be sent when making a request to `Url`.

**http_base_64**

The following functions are exported:

- `encode(PlainASCII)` -> `Base64`  
  [page 39] Encodes a plain ASCII string into base64.
- `decode(Base64)` -> `PlainASCII`  
  [page 39] Decodes an base64 encoded string to plain ASCII.

**httpd**

The following functions are exported:

- `start()`  
  [page 40] Start a server as specified in the given config file.
- `start(ConfigFile)` -> `ServerRet`  
  [page 40] Start a server as specified in the given config file.
- `start_link()`  
  [page 40] Start a server as specified in the given config file.
- `start_link(ConfigFile)` -> `ServerRet`  
  [page 40] Start a server as specified in the given config file.
- `restart()`  
  [page 40] Restart a running server.
- `restart(Port)` -> `ok | {error,Reason}`  
  [page 40] Restart a running server.
- `restart(ConfigFile)` -> `ok | {error,Reason}`  
  [page 40] Restart a running server.
- `restart(Address,Port)` -> `ok | {error,Reason}`  
  [page 40] Restart a running server.
- `stop()`  
  [page 41] Stop a running server.
- `stop(Port)` -> `ServerRet`  
  [page 41] Stop a running server.
- `stop(ConfigFile)` -> `ServerRet`  
  [page 41] Stop a running server.
- `stop(Address,Port)` -> `ServerRet`  
  [page 41] Stop a running server.
- `block()` -> `ok | {error,Reason}`  
  [page 41] Block a running server.
- `block(Port)` -> `ok | {error,Reason}`  
  [page 41] Block a running server.
- `block(ConfigFile)` -> `ok | {error,Reason}`  
  [page 41] Block a running server.
The following functions are exported:

- `check_enum(EnumString,ValidEnumStrings) -> Result` [page 45] Check if string is a valid enumeration.
- `clean(String) -> Stripped` [page 45] Remove leading and/or trailing white spaces.
- `custom_clean(String,Before,After) -> Stripped` [page 45] Remove leading and/or trailing white spaces and custom characters.
- `is_directory(FilePath) -> Result` [page 45] Check if a file path is a directory.
Inets Reference Manual

- is_file(FilePath) -> Result
  
  [page 46] Check if a file path is a regular file.

- make_integer(String) -> Result
  
  [page 46] Return an integer representation of a string.

httpd_core

No functions are exported.

httpd_socket

The following functions are exported:

- deliver(SocketType,Socket,Binary) -> Result
  
  [page 56] Send binary data over socket.

- peername(SocketType,Socket) -> {Port,IPAddress}
  
  [page 56] Return the port and IP-address of the remote socket.

- resolve() -> HostName
  
  [page 56] Return the official name of the current host.

httpd_util

The following functions are exported:

- convert_request_date(DateString) -> ErlDate|badDate
  
  [page 57] Convert the date to the Erlang date format.

- create_etag(FileInfo) -> Etag
  
  [page 57] Calculates the Etag for a file.

- decode_base64(Base64String) -> ASCIIString
  
  [page 57] Convert a base64 encoded string to a plain ascii string.

- decode_hex(HexValue) -> DecValue
  
  [page 57] Convert a hex value into its decimal equivalent.

- day(NthDayOfWeek) -> DayOfWeek
  
  [page 57] Convert the day of the week (integer [1-7]) to an abbreviated string.

- encode_base64(ASCIIString) -> Base64String
  
  [page 58] Convert an ASCII string to a Base64 encoded string.

- flatlength(NestedList) -> Size
  
  [page 58] Compute the size of a possibly nested list.

- header(StatusCode,PersistentConn)
  
  [page 58] Generate a HTTP 1.1 header.

- header(StatusCode,Date)
  
  [page 58] Generate a HTTP 1.1 header.

- header(StatusCode,MimeType,Date)
  
  [page 58] Generate a HTTP 1.1 header.

- header(StatusCode,MimeType,PersistentConn,Date) -> HTTPHeader
  
  [page 58] Generate a HTTP 1.1 header.
- `hexlist_to_integer(HexString) -> Number`
  [page 58] Convert a hexadecimal string to an integer.

- `integer_tohexlist(Number) -> HexString`
  [page 58] Convert an integer to a hexadecimal string.

- `keysearch(TupleList,Key)`
  [page 59] Search a list of key-value tuples for a tuple whose first element is a key.

- `keysearch(TupleList,Key,Undefined) -> Result`
  [page 59] Search a list of key-value tuples for a tuple whose first element is a key.

- `lookup(ETSTable,Key) -> Result`
  [page 59] Extract the first value associated with a key in an ETS table.

- `lookup(ETSTable,Key,Undefined) -> Result`
  [page 59] Extract the first value associated with a key in an ETS table.

- `lookup_mime(ConfigDB,Suffix) -> Result`
  [page 59] Return the mime type associated with a specific file suffix.

- `lookup_mime(ConfigDB,Suffix,Undefined) -> Result`
  [page 59] Return the mime type associated with a specific file suffix.

- `lookup_mime_default(ConfigDB,Suffix) -> Result`
  [page 59] Return the mime type associated with a specific file suffix or the value of
  the DefaultType.

- `lookup_mime_default(ConfigDB,Suffix,Undefined) -> Result`
  [page 59] Return the mime type associated with a specific file suffix or the value of
  the DefaultType.

- `message(StatusCode,PhraseArgs,ConfigDB) -> Message`
  [page 60] Return an informative HTTP 1.1 status string in HTML.

- `month(NthMonth) -> Month`
  [page 60] Convert the month as an integer (1-12) to an abbreviated string.

- `multi_lookup(ETSTable,Key) -> Result`
  [page 60] Extract the values associated with a key in a ETS table.

- `reason_phrase(StatusCode) -> Description`
  [page 60] Return the description of an HTTP 1.1 status code.

- `rfc1123_date() -> RFC1123Date`
  [page 61] Return the current date in RFC 1123 format.

- `rfc1123_date({'YYYY,MM,DD','Hour,Min,Sec'}) -> RFC1123Date`
  [page 61] Return the current date in RFC 1123 format.

- `split(String,RegExp,N) -> SplitRes`
  [page 61] Split a string in N chunks using a regular expression.

- `split_script_path(RequestLine) -> Splitted`
  [page 61] Split a RequestLine in a file reference to an executable and a
  queryString or a PathInfo String.

- `split_path(RequestLine) -> {Path,queryStringOrPathInfo}`
  [page 61] Split a RequestLine in a file reference and a queryString or a
  PathInfo String.

- `strip(String) -> Stripped`
  [page 61] Returns String where the leading and trailing space and tabs has been
  removed.

- `suffix(FileName) -> Suffix`
  [page 62] Extract the file suffix from a given filename.
mod_actions
No functions are exported.

mod_alias
The following functions are exported:

- **default_index(ConfigDB, Path)** -> **NewPath**
  - [page 66] Return a new path with the default resource or file appended.
- **path(Data, ConfigDB, RequestURI)** -> **Path**
  - [page 66] Return the actual file path to a URL.
- **real_name(ConfigDB, RequestURI, Aliases)** -> **Ret**
  - [page 67] Expand a request uri using Alias config directives.
- **real_script_name(ConfigDB, RequestURI, ScriptAliases)** -> **Ret**
  - [page 67] Expand a request uri using ScriptAlias config directives.

mod_auth
The following functions are exported:

- **add_user(UserName, Options)** -> **true | {error, Reason}**
  - [page 74] Add a user to the user database.
- **add_user(UserName, Password, UserData, Port, Dir)** -> **true | {error, Reason}**
  - [page 74] Add a user to the user database.
- **add_user(UserName, Password, UserData, Address, Port, Dir)** -> **true | {error, Reason}**
  - [page 74] Add a user to the user database.
- **delete_user(UserName, Options)** -> **true | {error, Reason}**
  - [page 74] Delete a user from the user database.
- **delete_user(UserName, Port, Dir)** -> **true | {error, Reason}**
  - [page 74] Delete a user from the user database.
- **delete_user(UserName, Address, Port, Dir)** -> **true | {error, Reason}**
  - [page 74] Delete a user from the user database.
- **get_user(UserName, Options)** -> **{ok, #httpd_user} | {error, Reason}**
  - [page 74] Returns a user from the user database.
- **get_user(UserName, Port, Dir)** -> **{ok, #httpd_user} | {error, Reason}**
  - [page 74] Returns a user from the user database.
- **get_user(UserName, Address, Port, Dir)** -> **{ok, #httpd_user} | {error, Reason}**
  - [page 74] Returns a user from the user database.
- `list_users(Options) -> {ok, Users} | {error, Reason}
  ![page 75] List users in the user database.
- `list_users(Address, Port, Dir) -> {ok, Users} | {error, Reason}
  ![page 75] List users in the user database.
- `add_group_member(GroupName, UserName, Options) -> true | {error, Reason}
  ![page 75] Add a user to a group.
- `add_group_member(GroupName, UserName, Port, Dir) -> true | {error, Reason}
  ![page 75] Add a user to a group.
- `add_group_member(GroupName, UserName, Address, Port, Dir) -> true | {error, Reason}
  ![page 75] Add a user to a group.
- `delete_group_member(GroupName, UserName, Options) -> true | {error, Reason}
  ![page 76] Remove a user from a group.
- `delete_group_member(GroupName, UserName, Port, Dir) -> true | {error, Reason}
  ![page 76] Remove a user from a group.
- `delete_group_member(GroupName, UserName, Address, Port, Dir) -> true | {error, Reason}
  ![page 76] Remove a user from a group.
- `list_group_members(GroupName, Options) -> {ok, Users} | {error, Reason}
  ![page 76] List the members of a group.
- `list_group_members(GroupName, Port, Dir) -> {ok, Users} | {error, Reason}
  ![page 76] List the members of a group.
- `list_group_members(GroupName, Address, Port, Dir) -> {ok, Users} | {error, Reason}
  ![page 76] List the members of a group.
- `list_groups(Options) -> {ok, Groups} | {error, Reason}
  ![page 76] List all the groups.
- `list_groups(Port, Dir) -> {ok, Groups} | {error, Reason}
  ![page 76] List all the groups.
- `list_groups(Address, Port, Dir) -> {ok, Groups} | {error, Reason}
  ![page 77] List all the groups.
- `delete_group(GroupName, Options) -> true | {error, Reason}
  ![page 77] Deletes a group
- `delete_group(GroupName, Address, Port, Dir) -> true | {error, Reason}
  ![page 77] Deletes a group
- `update_password(Port, Dir, OldPassword, NewPassword, NewPassword) -> ok | {error, Reason}
  ![page 77] Change the AuthAccessPassword
- `update_password(Address, Port, Dir, OldPassword, NewPassword, NewPassword) -> ok | {error, Reason}
  ![page 77] Change the AuthAccessPassword
mod_browser

The following functions are exported:

- `getBrowser(AgentString)` -> `{Browser,OperatingSystem}`  
  [page 79] Extracts the browser and operating-system from AgentString

mod_cgi

The following functions are exported:

- `env(Info,Script,AfterScript)` -> `EnvString`  
  [page 81] Return a CGI-1.1 environment variable string to be used by `open_port`.

- `status_code(CGIOutput)` -> `{ok,StatusCode} | {error,Reason}`  
  [page 81] Parse output from a CGI script and generates an appropriate HTTP status code.

mod_dir

No functions are exported.

mod_disk_log

The following functions are exported:

- `error_log(Socket,SocketType,ConfigDB,Date,Reason)` -> `ok | no_error_log`  
  [page 86] Log an error in the error log file.

- `security_log(User,Event)` -> `ok | no_security_log`  
  [page 87] Log an security event in the error log file.

mod_esi

The following functions are exported:

- `deliver(SessionID, Data)` -> `ok | {error,Reason}`  
  [page 92] Sends Data back to client.

- `Module:Function(Env, Input)` -> `Response`  
  [page 93] Creates a dynamic web page and return it as a list.

- `Module:Function(SessionID, Env, Input)` -> `void`  
  [page 93] Creates a dynamic web page and return it as a list.

mod_get

No functions are exported.

mod_head

No functions are exported.
mod_htaccess
No functions are exported.

mod_include
No functions are exported.

mod_log
The following functions are exported:

- `error_log(Socket,SocketType,ConfigDB,Date,Reason) -> ok | no_error_log`
  [page 106] Log an error in the a log file.

mod_range
No functions are exported.

mod_responsecontrol
No functions are exported.

mod_security
The following functions are exported:

- `list_auth_users(Port) -> Users | []`
  [page 111] List users that have authenticated within the SecurityAuthTimeout time for a given address (if specified), port number and directory (if specified).

- `list_auth_users(Address, Port) -> Users | []`
  [page 111] List users that have authenticated within the SecurityAuthTimeout time for a given address (if specified), port number and directory (if specified).

- `list_auth_users(Port, Dir) -> Users | []`
  [page 111] List users that have authenticated within the SecurityAuthTimeout time for a given address (if specified), port number and directory (if specified).

- `list_blocked_users(Port) -> Users | []`
  [page 111] List users that are currently blocked from access to a specified port number, for a given address (if specified).

- `list_blocked_users(Address, Port) -> Users | []`
  [page 111] List users that are currently blocked from access to a specified port number, for a given address (if specified).

- `list_blocked_users(Port, Dir) -> Users | []`
  [page 111] List users that are currently blocked from access to a specified port number, for a given address (if specified).
- `list_blocked_users(Address, Port, Dir) -> Users | []`
  [page 111] List users that are currently blocked from access to a specified port number, for a given address (if specified).

- `block_user(User, Port, Dir, Seconds) -> true | {error, Reason}`
  [page 112] Block user from access to a directory for a certain amount of time.

- `block_user(User, Address, Port, Dir, Seconds) -> true | {error, Reason}`
  [page 112] Block user from access to a directory for a certain amount of time.

- `unblock_user(User, Port) -> true | {error, Reason}`
  [page 112] Remove a blocked user from the block list.

- `unblock_user(User, Address, Port) -> true | {error, Reason}`
  [page 112] Remove a blocked user from the block list.

- `unblock_user(User, Port, Dir) -> true | {error, Reason}`
  [page 112] Remove a blocked user from the block list.

- `unblock_user(User, Address, Port, Dir) -> true | {error, Reason}`
  [page 112] Remove a blocked user from the block list.

- `event(What, Port, Dir, Data) -> ignored`
  [page 113] This function is called whenever an event occurs in mod_security.

- `event(What, Address, Port, Dir, Data) -> ignored`
  [page 113] This function is called whenever an event occurs in mod_security.

**mod_trace**

No functions are exported.
ftp
Erlang Module

The ftp module implements a client for file transfer according to a subset of the File Transfer Protocol (see RFC 959). Starting from inets version 4.4.1 the ftp client will always try to use passive ftp mode and only resort to active ftp mode if this fails. There is a mode option for open/[1,2,3] where this default behavior may be changed. Also the mode option replaces the API function force_active/1 that now has become deprecated. (force_active/1 is removed from the documentation but is still available in the code for now.)

For a simple example of an ftp session see Inets User's Guide. [page 2]
In addition to the ordinary functions for receiving and sending files (see recv/2, recv/3, send/2 and send/3) there are functions for receiving remote files as binaries (see recv_bin/2) and for sending binaries to to be stored as remote files (see send_bin/3).
There is also a set of functions for sending and receiving contiguous parts of a file to be stored in a remote file (for send see send_chunk_start/2, send_chunk/2 and send_chunk_end/1 and for receive see recv_chunk_start/2 and recv_chunk/).

The particular return values of the functions below depend very much on the implementation of the FTP server at the remote host. In particular the results from ls and nlist varies. Often real errors are not reported as errors by ls, even if for instance a file or directory does not exist. nlist is usually more strict, but some implementations have the peculiar behaviour of responding with an error, if the request is a listing of the contents of directory which exists but is empty.

COMMON DATA TYPES
Here follows type definitions that are used by more than one function in the FTP client API.

pid() - identifier of an ftp connection.
string() = list of ASCII characters
shortage_reason() = etnospc | epnospc
restriction_reason() = epath | efnamena | elogin | enotbinary
  - note not all restrictions may always relevant to all functions
common_reason() = econn | eclosed | term() - some kind of explanation of what went wrong
Exports

account(Pid, Account) -> ok | {error, Reason}

Types:
- Pid = pid()
- Account = string()
- Reason = eacct | common_reason()

If an account is needed for an operation set the account with this operation.

append(Pid, LocalFile [, RemoteFile]) -> ok | {error, Reason}

Types:
- Pid = pid()
- LocalFile = RemoteFile = string()
- Reason = epath | elogin | etnospc | epnospc | efnamena | common_reason()

Transfers the file LocalFile to the remote server. If RemoteFile is specified, the name of the remote file that the file will be appended to is set to RemoteFile; otherwise the name is set to LocalFile if the file does not exists the file will be created.

append_bin(Pid, Bin, RemoteFile) -> ok | {error, Reason}

Types:
- Pid = pid()
- Bin = binary()
- RemoteFile = string()
- Reason = restriction_reason() | shortage_reason() | common_reason()

Transfers the binary Bin to the remote server and append it to the file RemoteFile. If the file does not exists it will be created.

append_chunk(Pid, Bin) -> ok | {error, Reason}

Types:
- Pid = pid()
- Bin = binary()
- Reason = echunk | restriction_reason() | common_reason()

Transfer the chunk Bin to the remote server, which append it into the file specified in the call to append_chunk_start/2.

Note that for some errors, e.g. file system full, it is neccessary to to call append_chunk_end to get the proper reason.

append_chunk_start(Pid, File) -> ok | {error, Reason}

Types:
- Pid = pid()
- File = string()
- Reason = restriction_reason() | common_reason()
Start the transfer of chunks for appending to the file File at the remote server. If the file does not exist it will be created.

\[\text{append\_chunk\_end}(\text{Pid}) \rightarrow \text{ok} \mid \{\text{error, Reason}\}\]

Types:
- \text{Pid} = \text{pid()}
- \text{Reason} = \text{echunk} \mid \text{restriction\_reason()} \mid \text{shortage\_reason()}

Stops transfer of chunks for appending to the remote server. The file at the remote server, specified in the call to append\_chunk\_start/2 is closed by the server.

\[\text{cd}(\text{Pid}, \text{Dir}) \rightarrow \text{ok} \mid \{\text{error, Reason}\}\]

Types:
- \text{Pid} = \text{pid()}
- \text{Dir} = \text{string()}
- \text{Reason} = \text{restriction\_reason()} \mid \text{common\_reason()}

Changes the working directory at the remote server to Dir.

\[\text{close}(\text{Pid}) \rightarrow \text{ok}\]

Types:
- \text{Pid} = \text{pid()}

Ends the ftp session.

\[\text{delete}(\text{Pid, File}) \rightarrow \text{ok} \mid \{\text{error, Reason}\}\]

Types:
- \text{Pid} = \text{pid()}
- \text{File} = \text{string()}
- \text{Reason} = \text{restriction\_reason()} \mid \text{common\_reason()}

Deletes the file File at the remote server.

\[\text{formaterror}(\text{Tag}) \rightarrow \text{string()}\]

Types:
- \text{Tag} = \{\text{error, atom()}\} \mid \text{atom()}

Given an error return value \{error, AtomReason\}, this function returns a readable string describing the error.

\[\text{lcd}(\text{Pid, Dir}) \rightarrow \text{ok} \mid \{\text{error, Reason}\}\]

Types:
- \text{Pid} = \text{pid()}
- \text{Dir} = \text{string()}
- \text{Reason} = \text{restriction\_reason()}

Changes the working directory to Dir for the local client.

\[\text{lpwd}(\text{Pid}) \rightarrow \{\text{ok, Dir}\}\]
Types:
- Pid = pid()

Returns the current working directory at the local client.

\[ \text{ls(Pid [, Dir])} \rightarrow \{ \text{ok, Listing} \} \mid \{ \text{error, Reason} \} \]

Types:
- Pid = pid()
- Dir = string()
- Listing = string()
- Reason = restriction_reason() \mid \text{common_reason()}

Returns a listing of the contents of the remote current directory (\text{ls/1}) or the specified directory (\text{ls/2}). The format of Listing is operating system dependent (on UNIX it is typically produced from the output of the \text{ls -l} shell command).

\[ \text{mkdir(Pid, Dir)} \rightarrow \text{ok} \mid \{ \text{error, Reason} \} \]

Types:
- Pid = pid()
- Dir = string()
- Reason = restriction_reason() \mid \text{common_reason()}

Creates the directory Dir at the remote server.

\[ \text{nlist(Pid [, Dir])} \rightarrow \{ \text{ok, Listing} \} \mid \{ \text{error, Reason} \} \]

Types:
- Pid = pid()
- Dir = string()
- Listing = string()
- Reason = restriction_reason() \mid \text{common_reason()}

Returns a listing of the contents of the remote current directory (\text{nlist/1}) or the specified directory (\text{nlist/2}). The format of Listing is a stream of file names, where each name is separated by \text{CRLF} or \text{NL}. Contrary to the \text{ls} function, the purpose of \text{nlist} is to make it possible for a program to automatically process file name information.

\[ \text{open(Host [, Port] [, Flags])} \rightarrow \{ \text{ok, Pid} \} \mid \{ \text{error, Reason} \} \]

\[ \text{open(\{option_list, Option_list\})} \rightarrow \{ \text{ok, Pid} \} \mid \{ \text{error, Reason} \} \]

Types:
- Host = string() \mid \text{ip_address()}
- ip_address() = \{byte(), byte(), byte(), byte()\} \{byte(), byte(), byte(), byte(), byte(), byte(), byte(), byte()\}
- byte() = 0 | 1 | ... | 255
- Port = integer()
- Flags = [Flag]
- Flag = verbose \mid debug \mid ip_v6_disabled
- Pid = pid()
Inets Reference Manual

Reason = ehost
Option_list = [Options]
Options = [host, Host] | [port, Port] | [mode, Mode] | [flags, Flags] | [timeout, Timeout] | [progress, ProgressOption]
Mode = active | passive
Timeout = integer()
ProgressOption = ignore | [CBModule, CBFunction, InitProgress]
CallBackModule = atom()
CallBackFunction = atom()
InitProgress = term()

Opens a session with the ftp server at Host. The argument Host is either the name of the host, its IP address in dotted decimal notation (e.g. "150.236.14.136"), or a tuple of arity 4 (ipv4) or 8 (ipv6) (ex: [150, 236, 14, 136]).

If Port is supplied, a connection is attempted using this port number instead of the default (21).

Default value for Mode is passive.

If the atom verbose is included in Flags, response messages from the remote server will be written to standard output.

The progress option is intended to be used by applications that want to create some type of progress report such as a progress bar in a GUI. The default value for the progress option is ignore i.e. the option is not used. When the progress option is specified the following will happen when ftp:send/[3,4] or ftp:recv/[3,4] are called.

- Before a file is transferred the following call will be made to indicate the start of the file transfer and how big the file is. The return value of the callback function should be a new value for the UserProgressTerm that will be used as input next time the callback function is called.

  CBModule:CBFunction(InitProgress, File, {file_size, FileSize})

- Every time a chunk of bytes is transferred the following call will be made:

  CBModule:CBFunction(UserProgressTerm, File, {transfer_size, TransferSize})

- At the end of the file the following call will be made to indicate the end of the transfer.

  CBModule:CBFunction(UserProgressTerm, File, {transfer_size, 0})

The callback function should be defined as:

CBModule:CBFunction(UserProgressTerm, File, Size) -> UserProgressTerm

CBModule = CBFunction = atom()
UserProgressTerm = term()
File = string()
Size = [transfer_size, integer()] | [file_size, integer()] | [file_size, unknown]
Alas for remote files it is not possible for ftp to determine the file size in a platform independent way. In this case the size will be unknown and it is left to the application to find out the size.

**Note:**
The callback is made by a middleman process, hence the file transfer will not be affected by the code in the progress callback function. If the callback should crash this will be detected by the ftp connection process that will print an info-report and then go one as if the progress option was set to ignore.

The file transfer type is set to the default of the FTP server when the session is opened. This is usually ASCCI-mode.

The current local working directory (cf. `lpwd/1`) is set to the value reported by `file:get_cwd/1`, the wanted local directory.

The timeout value is default set to 60000 milliseconds.

The return value `Pid` is used as a reference to the newly created ftp client in all other functions, and they should be called by the process that created the connection. The ftp client process monitors the process that created it and will terminate if that process terminates.

```
pwd(Pid) -> {ok, Dir} | {error, Reason}
```

Types:
- `Pid = pid()`
- `Reason = restriction_reason() | common_reason()

Returns the current working directory at the remote server.

```
recv(Pid, RemoteFile [, LocalFile]) -> ok | {error, Reason}
```

Types:
- `Pid = pid()`
- `RemoteFile = LocalFile = string()`
- `Reason = restriction_reason() | common_reason() | file_write_error_reason()`
- `file_write_error_reason() = see file:write/2`

Transfer the file `RemoteFile` from the remote server to the the file system of the local client. If `LocalFile` is specified, the local file will be `LocalFile`; otherwise it will be `RemoteFile`.

If the file write fails (e.g. enospc), then the command is aborted and `{error, file_write_error_reason()}` is returned. The file is however not removed.

```
recv_bin(Pid, RemoteFile) -> {ok, Bin} | {error, Reason}
```

Types:
- `Pid = pid()`
- `Bin = binary()`
- `RemoteFile = string()`
- `Reason = restriction_reason() | common_reason()`
Transfers the file RemoteFile from the remote server and receives it as a binary.

\texttt{recv}\_\texttt{chunk}\_\texttt{start}(\texttt{Pid}, \texttt{RemoteFile}) \rightarrow \texttt{ok} | \{\texttt{error}, \texttt{Reason}\}

Types:
- \texttt{Pid} = \texttt{pid()}
- \texttt{RemoteFile} = \texttt{string()}
- \texttt{Reason} = \texttt{restriction\_reason() | common\_reason()}

Start transfer of the file \texttt{RemoteFile} from the remote server.

\texttt{recv}\_\texttt{chunk}(\texttt{Pid}) \rightarrow \texttt{ok} | \{\texttt{ok}, \texttt{Bin}\} | \{\texttt{error}, \texttt{Reason}\}

Types:
- \texttt{Pid} = \texttt{pid()}
- \texttt{Bin} = \texttt{binary()}
- \texttt{Reason} = \texttt{restriction\_reason() | common\_reason()}

Receive a chunk of the remote file (\texttt{RemoteFile} of \texttt{recv}\_\texttt{chunk}\_\texttt{start}). The return values has the following meaning:
- \texttt{ok} the transfer is complete.
- \{\texttt{ok}, \texttt{Bin}\} just another chunk of the file.
- \{\texttt{error}, \texttt{Reason}\} transfer failed.

\texttt{rename}(\texttt{Pid}, \texttt{Old}, \texttt{New}) \rightarrow \texttt{ok} | \{\texttt{error}, \texttt{Reason}\}

Types:
- \texttt{Pid} = \texttt{pid()}
- \texttt{CurrFile} = \texttt{NewFile} = \texttt{string()}
- \texttt{Reason} = \texttt{restriction\_reason() | common\_reason()}

 Renames \texttt{Old} to \texttt{New} at the remote server.

\texttt{rmdir}(\texttt{Pid}, \texttt{Dir}) \rightarrow \texttt{ok} | \{\texttt{error}, \texttt{Reason}\}

Types:
- \texttt{Pid} = \texttt{pid()}
- \texttt{Dir} = \texttt{string()}
- \texttt{Reason} = \texttt{restriction\_reason() | common\_reason()}

Removes directory \texttt{Dir} at the remote server.

\texttt{send}(\texttt{Pid}, \texttt{LocalFile} [, \texttt{RemoteFile}]) \rightarrow \texttt{ok} | \{\texttt{error}, \texttt{Reason}\}

Types:
- \texttt{Pid} = \texttt{pid()}
- \texttt{LocalFile} = \texttt{RemoteFile} = \texttt{string()}
- \texttt{Reason} = \texttt{restriction\_reason() | common\_reason() | shortage\_reason()}

Transfers the file \texttt{LocalFile} to the remote server. If \texttt{RemoteFile} is specified, the name of the remote file is set to \texttt{RemoteFile}; otherwise the name is set to \texttt{LocalFile}. 
send\_bin\(\text{Pid}, \text{Bin}, \text{RemoteFile}\) \(\rightarrow\) \(\text{ok} | \{\text{error, Reason}\}\)

Types:
- \(\text{Pid} = \text{pid()}\)
- \(\text{Bin} = \text{binary()}()\)
- \(\text{RemoteFile} = \text{string()}\)
- \(\text{Reason} = \text{restriction\_reason()} | \text{common\_reason()} | \text{shortage\_reason()}\)

Transfers the binary \(\text{Bin}\) into the file \(\text{RemoteFile}\) at the remote server.

send\_chunk\(\text{Pid}, \text{Bin}\) \(\rightarrow\) \(\text{ok} | \{\text{error, Reason}\}\)

Types:
- \(\text{Pid} = \text{pid()}\)
- \(\text{Bin} = \text{binary()}\)
- \(\text{Reason} = \text{echunk} | \text{restriction\_reason()} | \text{common\_reason()}\)

Transfer the chunk \(\text{Bin}\) to the remote server, which writes it into the file specified in the call to send\_chunk\_start/2.

Note that for some errors, e.g. file system full, it is neccessary to to call send\_chunk\_end to get the proper reason.

send\_chunk\_start\(\text{Pid}, \text{File}\) \(\rightarrow\) \(\text{ok} | \{\text{error, Reason}\}\)

Types:
- \(\text{Pid} = \text{pid()}\)
- \(\text{File} = \text{string()}\)
- \(\text{Reason} = \text{restriction\_reason()} | \text{common\_reason()}\)

Start transfer of chunks into the file \(\text{File}\) at the remote server.

send\_chunk\_end\(\text{Pid}\) \(\rightarrow\) \(\text{ok} | \{\text{error, Reason}\}\)

Types:
- \(\text{Pid} = \text{pid()}\)
- \(\text{Reason} = \text{restriction\_reason()} | \text{common\_reason()} | \text{shortage\_reason()}\)

Stops transfer of chunks to the remote server. The file at the remote server, specified in the call to send\_chunk\_start/2 is closed by the server.

type\(\text{Pid}, \text{Type}\) \(\rightarrow\) \(\text{ok} | \{\text{error, Reason}\}\)

Types:
- \(\text{Pid} = \text{pid()}\)
- \(\text{Type} = \text{ascii} | \text{binary}\)
- \(\text{Reason} = \text{etype} | \text{restriction\_reason()} | \text{common\_reason()}\)

Sets the file transfer type to ascii or binary. When an ftp session is opened, the transfer type is set to binary.

user\(\text{Pid}, \text{User}, \text{Password}\) \(\rightarrow\) \(\text{ok} | \{\text{error, Reason}\}\)

Types:
- \(\text{Pid} = \text{pid()}\)
• User = Password = string()
• Reason = euser | common_reason()

Performs login of User with Password.

user(Pid, User, Password, Account) -> ok | {error, Reason}

Types:
• Pid = pid()
• User = Password = string()
• Reason = euser | common_reason()

Performs login of User with Password to the account specified by Account.

quote(Pid, Command) -> [FTPLine]

Types:
• Pid = pid()
• Command = string()
• FTPLine = string() - Note the telnet end of line characters, from the ftp protocol definition, CRLF e.g. "\r\n" has been removed.

Sends an arbitrary FTP command and returns verbatimly a list of the lines sent back by the FTP server. This function is intended to give an application access to FTP commands that are server specific or that may not be provided by this FTP client.

**Note:**
FTP commands that require a data connection cannot be successfully issued with this function.

**ERRORS**

The possible error reasons and the corresponding diagnostic strings returned by formaterror/1 are as follows:

echunk Synchronisation error during chunk sending.
   A call has been made to send_chunk/2 or send_chunk_end/1, before a call to send_chunk_start/2; or a call has been made to another transfer function during chunk sending, i.e. before a call to send_chunk_end/1.

eclosed The session has been closed.
econn Connection to remote server prematurely closed.
ehost Host not found, FTP server not found, or connection rejected by FTP server.
elogin User not logged in.
ebinary Term is not a binary.
epath No such file or directory, or directory already exists, or permission denied.
etype No such type.
euser User name or password not valid.
etnospc Insufficient storage space in system [452].
epnospc Exceeded storage allocation (for current directory or dataset) [552].
efnamena File name not allowed [553].

SEE ALSO

This module provides the API to a HTTP/1.1 client according to RFC 2616, however this early version is somewhat limited for instance caching is not supported.

**Note:**
The functions `request/4`, and `set_options/1`, will start the inets application if it was not already started. When starting the inets application the client manager process that spawns request handlers, keeps track of proxy options etc will be started. Normally the application using this API should have started inets application. This is also true for the ssl application when using https. Also note that an application that does not set the pipeline-timeout value will benefit very little from pipelining as the default timeout is 0.

There are some usage examples in the Inets User's Guide. [page 2]

### COMMON DATA TYPES

Here follows type definitions that are used more than once in this module.

- `boolean() = true | false`
- `string() = list of ASCII characters`
- `request_id() = ref()`

### HTTP DATA TYPES

Here follows type definitions that are related to the HTTP protocol. For more information about the HTTP protocol see rfc 2616

- `method() = head | get | put | post | trace | options | delete`
- `request() - {url(), headers()} | {url(), headers(), content_type(), body()}
- `url() = string() - Syntax according to the URI definition in rfc 2396, ex: "http://www.example.com"
- `status_line() = {http_version(), status_code(), reason_phrase()}
- `http_version() = string() ex: "HTTP/1.1"
- `status_code() = integer()`
SSL DATA TYPES

Here follows some type definitions relevant when using https. for details [ssl(3)]

ssl_options() =
   {verify, code()} | {depth, depth()} | {certfile, path()}
   | {keyfile, path()} | {password, string()} | {cacertfile, path()}
   | {ciphers, string()}

Exports

cancel_request(RequestId) -> ok

Types:
   • RequestId = request_id() - A unique identifier as returned by request/4

Cancels an asynchronous HTTP-request.

request(Url) -> {ok, Result} | {error, Reason}

Types:
   • Url = url()
   • Result = {status_line(), headers(), body()} | {status_code(), body()} | request_id()
   • Reason = term()

Equivalent to http:request(get, {Url, []}, [], []).

request(Method, Request, HTTPOptions, Options) -> {ok, Result} | {error, Reason}

Types:
   • Method = method()
   • Request - request()
   • HTTPOptions - [HttpO ption]
   • HTTPOption - {timeout, integer()} | {ssl, ssl_options()} | {autoredirect, boolean()}

autoredirect - is true by default i.e. the client will automatically retrieve the information from the new URI and return that as the result instead of a 30X-result code. Note that for some 30X-result codes automatic redirect is not allowed in these cases the 30X-result will always be returned.
   • Options - {option()}
   • Option - {sync, boolean()} | {body_format, body_format()} | {full_result, boolean()}

The request function will be synchronous and return a full http response by default.
- `body_format()` = `string()` | `binary()`
  The `body_format` option is only valid for the synchronous request and the default is `string`, when making an asynchronous request the body will always be received as a binary.
- `Result = {status_line(), headers(), body()} | {status_code(), body()} | request_id()`
- `Reason = term()`

Sends a HTTP-request. The function can be both synchronous and asynchronous in the later case the function will return `{ok, RequestId}` and later on a message will be sent to the calling process on the format `{http, {RequestId, Result}}` or `{http, {RequestId, error, Reason}}`.

```erlang
set_options(Options) -> ok
```

**Types:**
- `Options = [Option]`
- `Option = {proxy, {Proxy, NoProxy}} | {max_sessions, MaxSessions} | {max_pipeline_length, MaxPipeline} | {pipeline_timeout, PipelineTimeout} | {cookies | CookieMode} | {ipv6, Ipv6Mode}`
- `Proxy = {Hostname, Port}`
  - `Hostname = string()`
  - `ex: "localhost" or "foo.bar.se"`
  - `Port = integer()`
  - `ex: 8080`
- `NoProxy = [NoProxyDesc]`
  - `NoProxyDesc = DomainDesc | HostName | IPDesc`
  - `DomainDesc = ".Domain"`
  - `ex: ".ericsson.se"`
  - `IpDesc = string()`
  - `ex: "134.138" or ["FEDC:BA98" (all IP-addresses starting with 134.138 or FEDC:BA98), "66.35.250.150" or ["2010:836B:4179::836B:4179"] (a complete IP-address).`
- `MaxSessions = integer()`
  - Maximum number of persistent connections to a host. Default is 2.
- `MaxPipeline = integer()`
  - Maximum number of outstanding requests on the same connection to a host. Default is 2.
- `PipelineTimeout = integer()`
  - If a persistent connection is idle longer than the pipeline_timeout the client will close the connection. Default is 0. The server may also have a such a time out but you should not count on it!
- `CookieMode = enabled | disabled | verify`
  - Default is disabled. If cookies are enabled all valid cookies will automatically be saved in the client managers cookie database. If the option verify is used the function `http:verify_cookie/2` has to be called for the cookie to be saved.
- `ipv6Mode = enabled | disabled`
  - By default enabled. This should normally be what you want. When it is enabled you can use both ipv4 and ipv6. The option is here to provide a workaround for buggy ipv6 stacks to ensure that ipv4 will always work.
Sets options to be used for subsequent requests. Later implementations may support user profiles, but currently these are global settings for all clients running on the same erlang node.

**Note:**
If possible the client will keep its connections alive and use them to pipeline requests whenever the circumstances allow. The HTTP/1.1 specification does not provide a guideline for how many requests that would be ideal to pipeline, this very much depends on the application. Note that a very long pipeline may cause a user perceived delays as earlier request may take a long time to complete. The HTTP/1.1 specification does suggest a limit of 2 persistent connections per server, which is the default value of the max_sessions option.

```erlang
verify_cookie(SetCookieHeaders, Url) -> ok

Types:
- SetCookieHeaders = headers() - where field = "set-cookie"
- Url = url()

Saves the cookies defined in SetCookieHeaders in the client managers cookie database. You need to call this function if you set the option cookies to verify.
```

```erlang
cookie_header(Url) -> header()

Types:
- Url = url()

Returns the cookie header that would be sent when making a request to Url.
```

**SEE ALSO**

RFC 2616, [ssl(3)]
http_base_64

Erlang Module

Implements base 64 encode and decode, see RFC 2045.

COMMON DATA TYPES

Here follows type definitions that are used by more than once this module.

string() = list of ASCII characters

Exports

encode(PlainASCII) -> Base64

Types:
• PlainASCII = string()
• Base64 = string()

Encodes a plain ASCII string into base64.

decode(Base64) -> PlainASCII

Types:
• PlainASCII = string()
• Base64 = string()

Decodes an base64 encoded string to plain ASCII.
httpd

Erlang Module

Exports

\[
\text{start()}
\]

\[
\text{start(ConfigFile) -> ServerRet}
\]

\[
\text{start_link()}
\]

\[
\text{start_link(ConfigFile) -> ServerRet}
\]

Types:
- ConfigFile = string()
- ServerRet = \{ok, Pid\} | ignore | \{error, Reason\} | \{stop, SReason\}
- Pid = pid()
- Reason = \{already_started, Pid\} | term()
- SReason = string()

\text{start/1 and start_link/1 starts a server as specified in the given ConfigFile. The ConfigFile supports a number of config directives specified below.}\n
\text{start/0 and start/0 starts a server as specified in a hard-wired config file, that is start("/var/tmp/server_root/conf/888.conf"). Before utilizing start/0 or start_link/0, copy the example server root\footnote{In Windows: `%INETS_ROOT%\examples\server_root`. In UNIX: `$INETS_ROOT/examples/server_root/`.} to a specific installation directory\footnote{In Windows: `X:\var\tmp`. In UNIX: `/var/tmp/`.} and you have a server running in no time.}\n
\text{If you copy the example server root to the specific installation directory it is furthermore easy to start an SSL enabled server, that is start("/var/tmp/server_root/conf/ssl.conf").}\n
\[
\text{restart()}
\]

\[
\text{restart(Port) -> ok | \{error, Reason\}}
\]

\[
\text{restart(ConfigFile) -> ok | \{error, Reason\}}
\]

\[
\text{restart(Address,Port) -> ok | \{error, Reason\}}
\]

Types:
- Port = integer()
- Address = \{A,B,C,D\} | string() | undefined
- ConfigFile = string()
- Reason = term()
restart restarts the server and reloads its config file.
The following directives cannot be changed: BindAddress, Port and SocketType. If these
should be changed, then a new server should be started instead.

**Note:**
Before the `restart` function can be called the server must be blocked [page 42].
After `restart` has been called, the server must be unblocked [page 42].

```
stop()  
stop(Port) -> ServerRet  
stop(ConfigFile) -> ServerRet  
stop(Address,Port) -> ServerRet

Types:
- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- ConfigFile = string()
- ServerRet = ok | not_started

stop/2 stops the server which listens to the specified Port on Address.
stop(integer()) stops a server which listens to a specific Port.
stop(string()) extracts BindAddress and Port from the config file and stops the server which listens to
the specified Port on Address. stop/0 stops a server which listens to port 8888, that is
stop(8888).

block() -> ok | {error,Reason}
block(Port) -> ok | {error,Reason}
block(ConfigFile) -> ok | {error,Reason}
block(Address,Port) -> ok | {error,Reason}
block(Port,Mode) -> ok | {error,Reason}
block(ConfigFile,Mode) -> ok | {error,Reason}
block(Address,Port,Mode) -> ok | {error,Reason}
block(ConfigFile,Mode,Timeout) -> ok | {error,Reason}
block(Address,Port,Mode,Timeout) -> ok | {error,Reason}

Types:
- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- ConfigFile = string()
- Mode = disturbing | non_disturbing
- Timeout = integer()
- Reason = term()
```
This function is used to block a server. The blocking can be done in two ways, disturbing or non-disturbing.

By performing a disturbing block, the server is blocked forcefully and all ongoing requests are terminated. If a timeout time is given then on-going requests are given this much time to complete before the server is forcefully blocked. In this case no new connections is accepted.

A non-disturbing block is more graceful. No new connections are accepted, but the ongoing requests are allowed to complete. If a timeout time is given, it waits this long before giving up (the block operation is aborted and the server state is once more not-blocked)

Default mode is disturbing.
Default port is 8888

unblock() -> ok | {error,Reason}
unblock(Port) -> ok | {error,Reason}
unblock(ConfigFile) -> ok | {error,Reason}
unblock(Address,Port) -> ok | {error,Reason}

Types:
- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- ConfigFile = string()
- Reason = term()

Unblocks a server. If the server is already unblocked this is a no-op. If a block is ongoing, then it is aborted (this will have no effect on ongoing requests).

parse_query(QueryString) -> ServerRet

Types:
- QueryString = string()
- ServerRet = [{Key,Value}]
- Key = Value = string()

parse_query/1 parses incoming data to erl and eval scripts (See mod_eis(3) [page 88]) as defined in the standard URL format, that is '+' becomes 'space' and decoding of hexadecimal characters (%xx).
ESWAPI CALLBACK FUNCTIONS

Exports

\[
\text{Module:do(Info)} \rightarrow \{\text{proceed,OldData}\} \mid \{\text{proceed,NewData}\} \mid \{\text{break,NewData}\} \mid \text{done}
\]

Types:
- \(\text{Info} = \text{mod}()\)
- \(\text{OldData} = \text{list}()\)
- \(\text{NewData} = \{\text{response,\{StatusCode,Body\}\}} \mid \{\text{response,\{response,Head,Body2\}\}} \mid \{\text{response,\{already_sent,Statuscode,Size\}\}}\)
- \(\text{Statuscode} = \text{integer}()\)
- \(\text{Body} = \text{String}\)
- \(\text{Head} = [\text{HeaderOption}]\)
- \(\text{HeaderOption} = \{\text{Key, Value}\} \mid \{\text{code, Statuscode}\}\)
- \(\text{Key} = \text{allow} \mid \text{cache_control} \mid \text{content_MD5} \mid \text{content_encoding} \mid \text{content_encoding} \mid \text{content_language} \mid \text{Value} \mid \text{content_length} \mid \text{content_location} \mid \text{content_range} \mid \text{content_type} \mid \text{date} \mid \text{etag} \mid \text{expires} \mid \text{last_modified} \mid \text{location} \mid \text{pragma} \mid \text{retry_after} \mid \text{server} \mid \text{trailer} \mid \text{transfer_encoding}\)
- \(\text{Value} = \text{string}()\)
- \(\text{Body2} = \{\text{Fun,Arg}\} \mid \text{Body} \mid \text{nobody}\)
- \(\text{Fun} = \text{fun( Arg)} \rightarrow \text{sent} \mid \text{close} \mid \text{Body}\)
- \(\text{Arg} = [\text{term()}]\)

\(\text{Info}\) is a record of type \text{mod}, this record is defined in \text{httpd.hrl} see EWSAPI Module programming in the Inets Users Guide [page 4] for more information.

When a valid request reaches \text{httpd} it calls \text{do}/1 in each module defined by the Modules configuration directive. The function may generate data for other modules or a response that can be sent back to the client.

The field \text{data} in \text{Info} is a list. This list will be the list returned from the from the last call to \text{do}/1.

\(\text{Body}\) is the body of the \text{http-response} that will be sent back to the client an appropriate header will be appended to the message. \(\text{Statuscode}\) will be the status code of the response see RFC2616 for the appropriate values.

\(\text{Head}\) is a key value list of HTTP header fields, the server will construct a HTTP header from this data. See RFC 2616 for the appropriate value for each header field. If the client is a HTTP/1.0 client then the server will filter the list so that only HTTP/1.0 header fields will be sent back to the client.

If \text{Body2} is returned and equal to \{\text{Fun,Arg}\} The Web server will try \text{apply}/2. on \text{Fun} with \text{Arg} as argument and expecxt that the fun either returns a list (\text{Body}) that is a HTTP-repsonse or the atom sent if the HTTP-response is sent back to the client. If close is returned from the fun something has gone wrong and the server will signal this to the client by closing the connection.
load/2 takes a row Line from the configuration file and tries to convert it to a key value tuple. If a directive is dependent on other directives, the directive may create a context. If the directive is not dependent on other directives return \{ok, [], Directive\}, otherwise return a new context, that is \{ok, NewContext\} or \{ok, Context Directive\}. If \{error, Reason\} is returned the configuration directive is assumed to be invalid.

Module:store([DirectiveKey, DirectiveValue], DirectiveList) -
\{ok, [DirectiveKey, NewDirectiveValue]\} | \{ok, [[ok, [DirectiveKey, NewDirectiveValue]] | \{error, Reason\}

Types:
- DirectiveKey = DirectiveValue = term()
- Directive = \{Key, Value\}
- Reason = term()

When all rows in the configuration file is read the function store/2 is called for each configuration directive. This makes it possible for a directive to alter other configuration directives. DirectiveList is a list of all configuration directives read in from load. If a directive may update other configuration directives then use this function.

Module:remove(ConfigDB) -
\{ok\} | \{error, Reason\}

Types:
- ConfigDB = ets_table()
- Reason = term()

When httpd shutdown it will try to execute remove/1 in each ewsapi module. The ewsapi programmer may use this to close ets tables, save data, or close down background processes.

SEE ALSO
httpd_core(3) [page 47], httpd_conf(3) [page 45], httpd_socket(3) [page 56], httpd_util(3) [page 57],
httpd_conf

Erlang Module

This module provides the EW SAPI programmer with utility functions for adding run-time configuration directives.

Warning:
The current implementation of EW SAPI is under review and feedback is welcomed.

Exports

check_enum(EnumString,ValidEnumStrings) -> Result

Types:
- EnumString = string()
- ValidEnumStrings = [string()]
- Result = {ok,atom() | {error,not_valid}}

check_enum/2 checks if EnumString is a valid enumeration of ValidEnumStrings in which case it is returned as an atom.

clean(String) -> Stripped

Types:
- String = Stripped = string()

clean/1 removes leading and/or trailing white spaces from String.

custom_clean(String,Before,After) -> Stripped

Types:
- Before = After = regexp()
- String = Stripped = string()

custom_clean/3 removes leading and/or trailing white spaces and custom characters from String. Before and After are regular expressions, as defined in regexp(3), describing the custom characters.

is_directory(FilePath) -> Result

Types:
- FilePath = string()
is_directory/1 checks if FilePath is a directory in which case it is returned. Please read file(3) for a description of enoent, eaccess and enotdir. The definition of the file info record can be found by including file.hrl from the kernel application, see file(3).

\[
is_{\text{directory}}(\text{FilePath}) \rightarrow \text{Result}
\]

Types:
- FilePath = string()
- Result = \{ok,Directory\} | \{error,Reason\}
- Directory = string()
- Reason = string() | enoent | eaccess | enotdir | FileInfo

FileInfo = File info record

is_file/1 checks if FilePath is a regular file in which case it is returned. Read file(3) for a description of enoent, eaccess and enotdir. The definition of the file info record can be found by including file.hrl from the kernel application, see file(3).

\[
is_{\text{file}}(\text{FilePath}) \rightarrow \text{Result}
\]

Types:
- FilePath = string()
- Result = \{ok,File\} | \{error,Reason\}
- File = string()
- Reason = string() | enoent | eaccess | enotdir | FileInfo

FileInfo = File info record

make_integer/1 returns an integer representation of String.

SEE ALSO

httpd(3) [page 40]
httpd_core

Erlang Module

This manual page summarize the core features of the server not being implemented as EWSAPI modules. The following core config directives are described:

Note:
There is no erlang module called httpd_core. This manual page is just a place to put the documentation for the core directives.

- BindAddress [page 48]
- DefaultType [page 49]
- DocumentRoot [page 49]
- MaxBodyAction [page 50]
- MaxBodySize [page 50]
- MaxClients [page 50]
- KeepAlive [page 49]
- KeepAliveTimeout [page 50]
- MaxHeaderAction [page 51]
- MaxHeaderSize [page 51]
- MaxKeepAliveRequests [page 51]
- Modules [page 51]
- Port [page 52]
- ServerAdmin [page 52]
- ServerName [page 52]
- ServerRoot [page 52]
- SocketType [page 53]
- SSLCACertificateFile [page 53]
- SSLCertificateFile [page 53]
- SSLCertificateKeyFile [page 53]
- SSLCiphers [page 54]
- SSLPasswordCallbackFunction [page 54]
- SSLPasswordCallbackModule [page 55]
- SSLVerifyClient [page 54]
- SSLVerifyDepth [page 54]
- DisableChunkedTransferEncodingSend [page 49]
SECURE SOCKETS LAYER (SSL)

The SSL support is realized using the SSL application based on OpenSSL. Please refer to [ssl(4)] for installation requirements for OpenSSL.

SSLeay is an implementation of Netscape's Secure Socket Layer specification - the software encryption protocol specification behind the Netscape Secure Server and the Netscape Navigator Browser.

The SSL Protocol can negotiate an encryption algorithm and session key as well as authenticate a server before the application protocol transmits or receives its first byte of data. All of the application protocol data is transmitted encrypted, ensuring privacy. The SSL protocol provides "channel security" which has three basic properties:

- The channel is private. Encryption is used for all messages after a simple handshake is used to define a secret key.
- The channel is authenticated. The server end-point of the conversation is always authenticated, while the client endpoint is optionally authenticated.
- The channel is reliable. The message transport includes a message integrity check (using a MAC).

The SSL mechanism can be enabled in the server by using the SSLCACertificateFile [page 53], SSLCertificateFile [page 53], SSLCertificateKeyFile [page 53], SSLCiphers [page 54], SSLVerifyDepth [page 54], and the SSLVerifyClient [page 54] config directives.

MIME TYPE SETTINGS

Files delivered to the client are MIME typed according to RFC 1590. File suffixes are mapped to MIME types before file delivery.

The mapping between file suffixes and MIME types are specified in the mime.types file. The mime.types reside within the conf directory of the ServerRoot [page 52]. Refer to the example server root. MIME types may be added as required to the mime.types file and the DefaultType [page 49] config directive can be used to specify a default mime type.

DIRECTIVE: "BindAddress"

Syntax: BindAddress address
Default: BindAddress *
Module: httpd_core(3) [page 47]

BindAddress defines which address the server will listen to. If the argument is * then the server listens to all addresses otherwise the server will only listen to the address specified. An address can be given either as an IP address or a hostname.

---

3In Windows: %INETS_ROOT%\examples\server_root. In UNIX: $INETS_ROOT/examples/server_root.
DIRECTIVE: "DefaultType"

Syntax: DefaultType mime-type
Default: -N o n e- 
Module: httpd_core(3) [page 47]

When the server is asked to provide a document type which cannot be determined by the MIME Type Settings [page 48], the server must inform the client about the content type of documents and mime-type is used if an unknown type is encountered.

DIRECTIVE: "DisableChunkedTransferEncodingSend"

Syntax: DisableChunkedTransferEncodingSend true | false
Default: false
Module: httpd_core(3) [page 47]

This directive tells the server whether to use chunked transfer-encoding when sending a response to a HTTP/1.1 client.

DIRECTIVE: "DocumentRoot"

Syntax: DocumentRoot directory-filename
Default: - M andatory -
Module: httpd_core(3) [page 47]

DocumentRoot points the Web server to the document space from which to serve documents from. Unless matched by a directive like Alias [page 65], the server appends the path from the requested URL to the DocumentRoot to make the path to the document, for example:

DocumentRoot /usr/web

and an access to http://your.server.org/index.html would refer to /usr/web/index.html.

DIRECTIVE: "KeepAlive"

Syntax: KeepAlive true | false
Default: true
Module: httpd_core(3) [page 47]

This directive tells the server whether to use persistent connection or not when the client claims to be HTTP/1.1 compliant. Note the value of KeepAlive has changed from previous versions to be compliant with Apache.
DIRECTIVE: "KeepAliveTimeout"

Syntax: KeepAliveTimeout seconds
Default: 150
Module: httpd_core(3) [page 47]

The number of seconds the server will wait for a subsequent request from the client before closing the connection. If the load on the server is high you may want to shorten this.

DIRECTIVE: "MaxBodyAction"

Syntax: MaxBodyAction action
Default: MaxBodyAction close Module: httpd_core(3) [page 47]

MaxBodyAction specifies the action to be taken when the message body limit has been passed.

- close the default and preferred communication type. ip_comm is also used for all remote message passing in Erlang.
- reply414 a reply (status) message with code 414 will be sent to the client prior to closing the socket. Note that this code is not defined in the HTTP/1.0 version of the protocol.

DIRECTIVE: "MaxBodySize"

Syntax: MaxBodySize size
Default: MaxBodySize nolimit Module: httpd_core(3) [page 47]

MaxBodySize limits the size of the message body of HTTP request. The reply to this is specified by the MaxBodyAction directive. Valid size is:

- nolimit the default message body limit, e.g. no limit.
- integer() any positive number.

DIRECTIVE: "MaxClients"

Syntax: MaxClients number
Default: MaxClients 150 Module: httpd_core(3) [page 47]

MaxClients limits the number of simultaneous requests that can be supported. No more than this number of child server processes can be created.
**DIRECTIVE: ”MaxHeaderAction”**

Syntax: MaxHeaderAction action
Default: MaxHeaderAction close Module: httpd_core(3) [page 47]

MaxHeaderAction specifies the action to be taken when the message Header limit has been passed.

- close: the socket is closed without any message to the client. This is the default action.
- reply414: a reply (status) message with code 414 will be sent to the client prior to closing the socket. Note that this code is not defined in the HTTP/1.0 version of the protocol.

**DIRECTIVE: ”MaxHeaderSize”**

Syntax: MaxHeaderSize size
Default: MaxHeaderSize 10240 Module: httpd_core(3) [page 47]

MaxHeaderSize limits the size of the message header of HTTP request. The reply to this is specified by the MaxHeaderAction directive. Valid size is:

- integer(): any positive number (default is 10240)
- nolimit: no limit should be applied

**DIRECTIVE: ”MaxKeepAliveRequests”**

Syntax: MaxKeepAliveRequests NumberOfRequests
Default: - Disabled - Module: httpd_core(3) [page 47]

The number of request that a client can do on one connection. When the server has responded to the number of requests defined by MaxKeepAliveRequests the server close the connection. The server will close it even if there are queued request.

**DIRECTIVE: ”Modules”**

Syntax: Modules module module ...
Default: Modules mod_set mod_head mod_log Module: httpd_core(3) [page 47]

Modules defines which EWSAPI modules to be used in a specific server setup. module is a module in the code path of the server which has been written in accordance with the Erlang Web Server API in the Inets Users Guide. [page 4] The server executes functionality in each module, from left to right (from now on called EWSAPI Module Sequence).

Before altering the EWSAPI Modules Sequence please observe what types of data each module uses and propagates. Read the “EWSAPI Interaction” section for each module and the EWSAPI Module Interaction in the Inets Users Guide [page 4] description in httpd(3).
DIRECTIVE: "Port"

Syntax: Port number
Default: Port 80
Module: httpd_core(3) [page 47]

Port defines which port number the server should use (0 to 65535). Certain port numbers are reserved for particular protocols, i.e. examine your OS characteristics\(^4\) for a list of reserved ports. The standard port for HTTP is 80.

All ports numbered below 1024 are reserved for system use and regular (non-root) users cannot use them, i.e. to use port 80 you must start the Erlang node as root. (sic!) If you do not have root access choose an unused port above 1024 typically 8000, 8080 or 8888.

DIRECTIVE: "ServerAdmin"

Syntax: ServerAdmin email-address
Default: ServerAdmin unknown@unknown
Module: httpd_core(3) [page 47]

ServerAdmin defines the email-address of the server administrator, to be included in any error messages returned by the server. It may be worth setting up a dedicated user for this because clients do not always state which server they have comments about, for example:

ServerAdmin www-admin@white-house.com

DIRECTIVE: "ServerName"

Syntax: ServerName fully-qualified domain name
Default: - Mandatory -
Module: httpd_core(3) [page 47]

ServerName sets the fully-qualified domain name of the server.

DIRECTIVE: "ServerRoot"

Syntax: ServerRoot directory-filename
Default: - Mandatory -
Module: httpd_core(3) [page 47]

ServerRoot defines a directory-filename where the server has it's operational home, e.g. used to store log files and system icons. Relative paths specified in the config file refer to this directory-filename (See mod_log(3) [page 104]).

\(^4\)In UNIX: /etc/services.
 DIRECTIVE: "SocketType"

Syntax: SocketType type
Default: SocketType ip_comm
Module: httpd_core(3) [page 47]

SocketType defines which underlying communication type to be used. Valid socket types are:

- ip Comm: the default and preferred communication type. ip Comm is also used for all remote message passing in Erlang.
- ssl: the communication type to be used to support SSL (Read more about Secure Sockets Layer (SSL) [page 48] in httpd(3)).

 DIRECTIVE: "SSLCACertificateFile"

Syntax: SSLCACertificateFile filename
Default: - None -
Module: httpd_core(3) [page 47]

SSLCACertificateFile points at a PEM encoded certificate of the certification authorities. Read more about PEM encoded certificates in the SSL application documentation. Read more about PEM encoded certificates in the SSL application documentation.

 DIRECTIVE: "SSLCertificateFile"

Syntax: SSLCertificateFile filename
Default: - None -
Module: httpd_core(3) [page 47]

SSLCertificateFile points at a PEM encoded certificate. Read more about PEM encoded certificates in the SSL application documentation. The dummy certificate server.pem, in the Inets distribution, can be used for test purposes. Read more about PEM encoded certificates in the SSL application documentation.

 DIRECTIVE: "SSLCertificateKeyFile"

Syntax: SSLCertificateKeyFile filename
Default: - None -
Module: httpd_core(3) [page 47]

SSLCertificateKeyFile is used to point at a certificate key file. This directive should only be used if a certificate key has not been bundled with the certificate file pointed at by SSLCertificateFile [page 53].

5In Windows: %INETS%\examples\server_root\ssl\ In UNIX: $INETS/examples/server_root/ssl/.
DIRECTIVE: "SSLVerifyClient"

Syntax: SSLVerifyClient type
Default: - None -
Module: httpd_core(3) [page 47]
Set type to:

0 if no client certificate is required.
1 if the client may present a valid certificate.
2 if the client must present a valid certificate.
3 if the client may present a valid certificate but it is not required to have a valid CA.

Read more about SSL in the application documentation.

DIRECTIVE: "SSLVerifyDepth"

Syntax: SSLVerifyDepth integer
Default: - None -
Module: httpd_core(3) [page 47]
This directive specifies how far up or down the (certification) chain we are prepared to go before giving up.
Read more about SSL in the application documentation.

DIRECTIVE: "SSLCiphers"

Syntax: SSLCiphers ciphers
Default: - None -
Module: httpd_core(3) [page 47]
SSLCiphers is a colon separated list of ciphers.
Read more about SSL in the application documentation.

DIRECTIVE: "SSLPasswordCallbackFunction"

Syntax: SSLPasswordCallbackFunction function
Default: - None -
Module: httpd_core(3) [page 47]
The SSLPasswordCallbackFunction function in module SSLPasswordCallbackModule is called in order to retrieve the user's password.
Read more about SSL in the application documentation.
DIRECTIVE: "SSLPasswordCallbackModule"

Syntax: SSLPasswordCallbackModule function
Default: -None-
Module: httpd_core(3) [page 47]

The SSLPasswordCallbackFunction function in the SSLPasswordCallbackModule module is called in order to retrieve the user's password. Read more about SSL in the application documentation.

SEE ALSO

httpd(3) [page 40]
httpd_socket

Erlang Module

This module provides the EWSAPI module programmer with utility functions for generic sockets communication. The appropriate communication mechanism is transparently used, that is ip_comm or ssl.

Exports

deliver(SocketType,Socket,Binary) -> Result

Types:
- SocketType = socket_type()
- Socket = socket()
- Binary = binary()
- Result = socket_closed | void()

deliver/3 sends the Binary over the Socket using the specified SocketType. Socket and SocketType should be the socket and the socket_type form the mod record as defined in httpd.

peername(SocketType,Socket) -> {Port,IPAddress}

Types:
- SocketType = socket_type()
- Socket = socket()
- Port = integer()
- IPAddress = string()

peername/3 returns the Port and IPAddress of the remote Socket.

resolve() -> HostName

Types:
- HostName = string()

resolve/0 returns the official HostName of the current host.

SEE ALSO

httpd(3) [page 40]
httpd_util

Erlang Module

This module provides the Erlang Web Server API in the Inets Users Guide [page 4] module programmer with miscellaneous utility functions.

Exports

\[\text{convert_request_date(DateString) -> ErlDate|bad} \]

Types:
- DateString = string()
- ErlDate = [Year,Month,Date],[Hour,Min,Sec]
- Year = Month = Date = Hour = Min = Sec = integer()

\[\text{convert_request_date/1 converts } \text{DateString to the Erlang date format. DateString must be in one of the three date formats that is defined in the RFC 2616.} \]

\[\text{create_etag(FileInfo) -> Etag} \]

Types:
- FileInfo = file_info()
- Etag = string()

create_etag/1 calculates the Etag for a file, from it’s size and time for last modification. fileinfo is a record defined in kernel/include/file.hrl

\[\text{decode_base64(Base64String) -> ASCIIString} \]

Types:
- Base64String = ASCIIString = string()

Deprecated use \text{http_base64:decode/1} [page 39]

\[\text{decode_hex(HexValue) -> DecValue} \]

Types:
- HexValue = DecValue = string()

Converts the hexadecimal value HexValue into it’s decimal equivalent (DecValue).

\[\text{day(NthDayOfWeek) -> DayOfWeek} \]

Types:
- NthDayOfWeek = 1-7
- DayOfWeek = string()
day/1 converts the day of the week (NthDayOfWeek) as an integer (1-7) to an abbreviated string, that is:
1 = “Mon”, 2 = “Tue”, ..., 7 = “Sat”.

```
encode_base64(ASCIIString) -> Base64String
```

Types:
- ASCIIString = string()
- Base64String = string()

Deprecated use http_base64:decode/1 [page 39]

```
flatlength(NestedList) -> Size
```

Types:
- NestedList = list()
- Size = integer()

flatlength/1 computes the size of the possibly nested list NestedList. Which may contain binaries.

```
header(StatusCode, PersistentConn)
header(StatusCode, Date)
header(StatusCode, MimeType, Date)
header(StatusCode, MimeType, PersistentConn, Date) -> HTTPHeader
```

Types:
- StatusCode = integer()
- Date = rfc1123_date()
- MimeType = string()
- PersistentConn = true | false

header returns a HTTP 1.1 header string. The StatusCode is one of the status codes defined in RFC 2616 and the Date string is RFC 1123 compliant. (See rfc1123_date/0 [page 61]).

Note that the two version of header/n that does not has a PersistentConn argument is there only for backward compability, and must not be used in new EWSAPI modules that will support persistent connections.

```
hexlist_to_integer(HexString) -> Number
```

Types:
- Number = integer()
- HexString = string()

hexlist_to_integer Convert the Hexadecimal value of HexString to an integer.

```
integer_tohexlist(Number) -> HexString
```

Types:
- Number = integer()
- HexString = string()
integer_to_hexlist/1 Returns a string that represents the number in a hexadecimal form.

key1search(TupleList,Key)
key1search(TupleList,Key,Undefined) -> Result
Types:
- TupleList = [tuple()]
- Key = term()
- Result = term() | undefined | Undefined
- Undefined = term()
key1search searches the TupleList for a tuple whose first element is Key.
key1search/2 returns undefined and key1search/3 returns Undefined if no tuple is found.

lookup(ETSTable,Key) -> Result
lookup(ETSTable,Key,Undefined) -> Result
Types:
- ETSTable = ets_table()
- Key = term()
- Result = term() | undefined | Undefined
- Undefined = term()
lookup extracts {Key,Value} tuples from ETSTable and returns the Value associated with Key. If ETSTable is of type bag only the first Value associated with Key is returned.
lookup/2 returns undefined and lookup/3 returns Undefined if no Value is found.

lookup_mime(ConfigDB,Suffix)
lookup_mime(ConfigDB,Suffix,Undefined) -> MimeType
Types:
- ConfigDB = ets_table()
- Suffix = string()
- MimeType = string() | undefined | Undefined
- Undefined = term()
lookup_mime returns the mime type associated with a specific file suffix as specified in the mime.types file (located in the config directory\(^6\)).

lookup_mime_default(ConfigDB,Suffix)
lookup_mime_default(ConfigDB,Suffix,Undefined) -> MimeType
Types:
- ConfigDB = ets_table()
- Suffix = string()
- MimeType = string() | undefined | Undefined
- Undefined = term()
lookup_mime_default returns the mime type associated with a specific file suffix as specified in the mime.types file (located in the config directory\(^6\)).

\(^6\)In Windows: %SERVER_ROOT%\conf\mime.types. In UNIX: $SERVER_ROOT/conf/mime.types.
lookup_mime_default returns the mime type associated with a specific file suffix as specified in the mime.types file (located in the config directory\(^7\)). If no appropriate association can be found the value of DefaultType [page 49] is returned.

message(StatusCode,PhraseArgs,ConfigDB) -> Message

Types:
- StatusCode = 301 | 400 | 403 | 404 | 500 | 501 | 504
- PhraseArgs = term()
- ConfigDB = ets_table
- Message = string()

message/3 returns an informative HTTP 1.1 status string in HTML. Each StatusCode requires a specific PhraseArgs:

- 301 string(): A URL pointing at the new document position.
- 400 | 401 | 500 none (No PhraseArgs)
- 403 | 404 string(): A Request-URI as described in RFC 2616.
- 504 string(): A string describing why the service was unavailable.

month(NthMonth) -> Month

Types:
- NthMonth = 1-12
- Month = string()

month/1 converts the month NthMonth as an integer (1-12) to an abbreviated string, that is:

1 = “Jan”, 2 = “Feb”, ..., 12 = “Dec”.

multi_lookup(ETSTable,Key) -> Result

Types:
- ETSTable = ets_table()
- Key = term()
- Result = [term()]

multi_lookup extracts all {Key,Value} tuples from an ETSTable and returns all Values associated with the Key in a list.

reason_phrase(StatusCode) -> Description

Types:
- StatusCode = 100| 200 | 201 | 202 | 204 | 205 | 206 | 300 | 301 | 302 | 303 | 304 | 400 | 401 | 402 | 403 | 404 | 405 | 406 | 410 411 | 412 | 413 | 414 415 | 416 | 417 | 500 | 501 | 502 | 503 | 504 | 505
- Description = string()

\(^7\)In Windows: %SERVER\_ROOT%\conf\mime.types. In UNIX: $SERVER\_ROOT/conf/mime.types.
reason_phrase returns the Description of an HTTP 1.1 StatusCode, for example 200 is "OK" and 201 is "Created". Read RFC 2616 for further information.

rfc1123_date() -> RFC1123Date
rfc1123_date([YYYY,MM,DD],[Hour,Min,Sec]) -> RFC1123Date

Types:
- YYYY = MM = DD = Hour = Min = Sec = integer()
- RFC 1123Date = string()

rfc1123_date/0 returns the current date in RFC 1123 format. rfc_date/1 converts the date in the Erlang format to the RFC 1123 date format.

split(String,RegExp,N) -> SplitRes

Types:
- String = RegExp = string()
- SplitRes = {ok, FieldList} | {error, errordesc()}
- Fieldlist = [string()]
- N = integer

split/3 splits the String in N chunks using the RegExp. split/3 is is equivalent to regexp:split/2 with one exception, that is N defines the number of maximum number of fields in the FieldList.

split_script_path(RequestLine) -> Splitted

Types:
- RequestLine = string()
- Splitted = not_a_script | {Path, PathInfo, QueryString}
- Path = QueryString = PathInfo = string()

split_script_path/1 is equivalent to split_path/1 with one exception. If the longest possible path is not a regular, accessible and executable file not_a_script is returned.

split_path(RequestLine) -> {Path,QueryStringOrPathInfo}

Types:
- RequestLine = Path = QueryStringOrPathInfo = string()

split_path/1 splits the RequestLine in a file reference (Path) and a QueryString or a PathInfo string as specified in RFC 2616. A QueryString is isolated from the Path with a question mark (?) and PathInfo with a slash (/). In the case of a QueryString, everything before the ? is a Path and everything after a QueryString. In the case of a PathInfo the RequestLine is scanned from left-to-right on the hunt for longest possible Path being a file or a directory. Everything after the longest possible Path, isolated with a /, is regarded as PathInfo. The resulting Path is decoded using decode_hex/1 before delivery.

strip(String) -> Stripped

Types:
- String = Stripped = string()
strip/1 removes any leading or trailing linear white space from the string. Linear white space should be read as horizontal tab or space.

suffix(FileNamé) -> Suffix

Types:
- FileName = Suffix = string()
suffix/1 is equivalent to filename:extension/1 with one exception, that is Suffix is returned without a leading dot (.)

tolower(String) -> ConvertedString

Types:
- String = ConvertedString = string()
tolower/1 converts upper-case letters to lower-case.

toupper(String) -> ConvertedString

Types:
- String = ConvertedString = string()
toupper/1 converts lower-case letters to upper-case.

SEE ALSO

httpd(3) [page 40]
mod_actions

Erlang Module

This module runs CGI scripts whenever a file of a certain type or HTTP method (See RFC 1945) is requested. The following config directives are described:

- Action [page 63]
- Script [page 63]

DIRECTIVE: "Action"

Syntax: Action mime-type cgi-script
Default: - None -
Module: mod_actions(3) [page 63]

Action adds an action, which will activate a cgi-script whenever a file of a certain mime-type is requested. It propagates the URL and file path of the requested document using the standard CGI PATH_INFO and PATH_TRANSLATED environment variables.

Examples:
Action text/plain /cgi-bin/log_and_deliver.txt
Action home-grown/mime-type1 ~/bob/do_special_stuff

DIRECTIVE: "Script"

Syntax: Script method cgi-script
Default: - None -
Module: mod_actions(3) [page 63]

Script adds an action, which will activate a cgi-script whenever a file is requested using a certain HTTP method. The method is either GET or POST as defined in RFC 1945. It propagates the URL and file path of the requested document using the standard CGI PATH_INFO and PATH_TRANSLATED environment variables.

Examples:
Script GET /cgi-bin/get
Script POST ~/bob/put_and_a_little_more
EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:

{real_name,{Path,AfterPath}} as defined in mod_alias(3) [page 65].

Exports the following EWSAPI interaction data, if possible:

{new_request_uri,RequestURI} An alternative RequestURI has been generated.

Uses the following exported EWSAPI functions:

- mod_alias:path/3 [page 67]

SEE ALSO

httpd(3) [page 40], mod_alias(3) [page 65]
mod_alias
Erlang Module

This module makes it possible to map different parts of the host file system into the document tree. The following config directives are described:

- Alias [page 65]
- DirectoryIndex [page 65]
- ScriptAlias [page 66]

DIRECTIVE: ”Alias”

Syntax: Alias url-path directory-filename
Default: - None -
Module: mod_alias(3) [page 65]

The Alias directive allows documents to be stored in the local file system instead of the DocumentRoot [page 49] location. URLs with a path that begins with url-path is mapped to local files that begins with directory-filename, for example:

```
Alias /image /ftp/pub/image
```

and an access to http://your.server.org/image/foo.gif would refer to the file /ftp/pub/image/foo.gif.

DIRECTIVE: ”DirectoryIndex”

Syntax: DirectoryIndex file file ...
Default: - None -
Module: mod_alias(3) [page 65]

DirectoryIndex specifies a list of resources to look for if a client requests a directory using a / at the end of the directory name. file depicts the name of a file in the directory. Several files may be given, in which case the server will return the first it finds, for example:

```
DirectoryIndex index.html
```

and access to http://your.server.org/docs/ would return http://your.server.org/docs/index.html if it existed.
DIRECTIVE: "ScriptAlias"

Syntax: ScriptAlias url-path directory-filename
Default: -None-
Module: mod_alias(3) [page 65]
The ScriptAlias directive has the same behavior as the Alias [page 65] directive, except
that it also marks the target directory as containing CGI scripts. URLs with a path
beginning with url-path are mapped to scripts beginning with directory-filename,
for example:
ScriptAlias /cgi-bin/ /web/cgi-bin/

and an access to http://your.server.org/cgi-bin/foo would cause the server to run
the script /web/cgi-bin/foo.

EWSAPI MODULE INTERACTION

Exports the following EW SAPI interaction data, if possible:
{real_name, {Path, AfterPath}} Path and AfterPath is as defined in
httpd_util:split_path/1 [page 61] with one exception - Path has been run through
default_index/2 [page 66].

Uses the following exported EW SAPI functions:
• mod_alias:default_index/2 [page 66]
• mod_alias:path/3 [page 67]
• mod_alias:real_name/3 [page 67]

This module furthermore exports a batch of functions to be used by other EW SAPI
modules:

Exports
default_index(ConfigDB, Path) -> NewPath

Types:
• ConfigDB = config_db()
• Path = NewPath = string()

If Path is a directory, default_index/2, it starts searching for resources or files that are
specified in the config directive DirectoryIndex [page 65]. If an appropriate resource or
file is found, it is appended to the end of Path and then returned. Path is returned
unaltered, if no appropriate file is found, or if Path is not a directory. config_db() is
the server config file in ETS table format as described in EW SAPI Module Programming
in the Inets Users Guide. [page 4].

path(Data, ConfigDB, RequestURI) -> Path

Types:
- $\text{Data} = \text{interaction.data()}$
- $\text{ConfigDB} = \text{config.db()}$
- $\text{RequestURI} = \text{Path} = \text{string()}$

path/3 returns the actual file Path in the RequestURI (See RFC 1945). If the interaction data \{real.name, \{Path, AfterPath\}\} has been exported by mod.alias(3) [page 66]; Path is returned. If no interaction data has been exported, ServerRoot [page 52] is used to generate a file Path. config.db() and interaction.data() are as defined in EWSAPI Module Programming in the Inets Users Guide [page 4].

\[
\text{real.name(ConfigDB, RequestURI, Aliases) -> Ret}
\]

Types:
- $\text{ConfigDB} = \text{config.db()}$
- $\text{RequestURI} = \text{string()}$
- $\text{Aliases} = \{\text{FakeName, RealName}\}$
- $\text{Ret} = \{\text{ShortPath, Path, AfterPath}\}$
- $\text{ShortPath} = \text{Path} = \text{AfterPath} = \text{string()}$

\text{real.name/3} traverses Aliases, typically extracted from ConfigDB, and matches each FakeName with RequestURI. If a match is found FakeName is replaced with RealName in the match. The resulting path is split into two parts, that is ShortPath and AfterPath as defined in httpd.util:split.path/1 [page 61]. Path is generated from ShortPath, that is the result from default_index/2 [page 66] with ShortPath as an argument.

config.db() is the server config file in ETS table format as described in EWSAPI Module Programming in the Inets Users Guide. [page 4].

\[
\text{real_script.name(ConfigDB, RequestURI, ScriptAliases) -> Ret}
\]

Types:
- $\text{ConfigDB} = \text{config.db()}$
- $\text{RequestURI} = \text{string()}$
- $\text{ScriptAliases} = \{\text{FakeName, RealName}\}$
- $\text{Ret} = \{\text{ShortPath, AfterPath} \mid \text{not a script}\}$
- $\text{ShortPath} = \text{AfterPath} = \text{string()}$

real_script.name/3 traverses ScriptAliases, typically extracted from ConfigDB, and matches each FakeName with RequestURI. If a match is found FakeName is replaced with RealName in the match. If the resulting match is not an executable script \text{not a script} is returned. If it is a script the resulting script path is in two parts, that is ShortPath and AfterPath as defined in httpd.util:split.script.path/1 [page 61]. config.db() is the server config file in ETS table format as described in EWSAPI Module Programming in the Inets Users Guide. [page 4].

\text{SEE ALSO}

httpd(3) [page 40]
mod_auth

Erlang Module

This module provides for basic user authentication using textual files, dets databases as well as mnesia databases. The following config directives are supported:

- `<Directory>` [page 68]
- `AuthDBType` [page 69]
- `AuthAccessPassword` [page 72]
- `AuthUserFile` [page 70]
- `AuthGroupFile` [page 71]
- `AuthName` [page 71]
- `allow` [page 72]
- `deny` [page 72]
- `require` [page 73]

The `Directory` [page 68] config directive is central to be able to restrict access to certain areas of the server. Please read about the `Directory` [page 68] config directive.

**DIRECTIVE: "Directory"**

Syntax: `<Directory regexp-filename>`

Default: - None -

Module: mod_auth(3) [page 68]

Related: `allow` [page 72], `deny` [page 72], `AuthAccessPassword` [page 72] `AuthUserFile` [page 70], `AuthGroupFile` [page 71], `AuthName` [page 71], `require` [page 73]

`<Directory>` and `/<Directory>` are used to enclose a group of directives which applies only to the named directory and sub-directories of that directory.

`regexp-filename` is an extended regular expression (See `regexp(3)`). For example:

```
<Directory /usr/local/httpd[12]/htdocs>
  AuthAccessPassword sOmePassWorD
  AuthDBType plain
  AuthName My Secret Garden
  AuthUserFile /var/tmp/server.root/auth/user
  AuthGroupFile /var/tmp/server.root/auth/group
  require user ragnar edward
  require group group1
  allow from 123.145.244.5
</Directory>
```

If multiple directory sections match the directory (or its parents), then the directives are applied with the shortest match first. For example if you have one directory section for garden/ and one for garden/flowers, the garden/ section matches first.
DIRECTIVE: "AuthDBType"

Syntax: AuthDBType plain | dets | mnesia

Default: -None-

Module: mod_auth(3) [page 68]

Context: <Directory> [page 68]

Related: allow [page 72], deny [page 72], AuthAccessPassword [page 72], AuthName [page 71], AuthUserFile [page 70], AuthGroupFile [page 71], require [page 73]

AuthDBType sets the type of authentication database that is used for the directory. The key difference between the different methods is that dynamic data can be saved when Mnesia and Dets is used.

If Mnesia is used as storage method, Mnesia must be started prior to the webserver. The first time Mnesia is started the schema and the tables must be created before Mnesia is started. A naive example of a module with two functions that creates and start mnesia is provided here. The function shall be sued the first time. first_start/0 creates the schema and the tables. The second function start/0 shall be used in consecutive startups. start/0 starts Mnesia and wait for the tables to be initiated. This function must only be used when the schema and the tables already is created.

```erlang
-module(mnesia_test).
-export([start/0,load_data/0]).
-include("mod_auth.hrl").

first_start()->
  mnesia:create_schema([node()]),
  mnesia:start(),
  mnesia:create_table(httpd_user,
    [{type,bag},{disc_copies,[node()]},
      {attributes,record_info(fields,httpd_user)}]),
  mnesia:create_table(httpd_group,
    [{type,bag},{disc_copies,[node()]},
      {attributes,record_info(fields,httpd_group)}]),
  mnesia:wait_for_tables([httpd_user,httpd_group],60000).

start()->
  mnesia:start(),
  mnesia:wait_for_tables([httpd_user,httpd_group],60000).
```

To create the Mnesia tables we use two records defined in mod_auth.hrl so the file must be included.

The first function first_start/0 creates a schema that specify on which nodes the database shall reside. Then it starts Mnesia and creates the tables. The first argument is the name of the tables, the second argument is a list of options how the table will be created, see Mnesia documentation for more information. Since the current implementation of the mod_auth_mnesia saves one row for each user the type must be bag.

When the schema and the tables is created the second function start/0 shall be used to start Mnesia. It starts Mnesia and wait for the tables to be loaded. Mnesia use the directory specified as mnesia_dir at startup if specified, otherwise Mnesia use the current directory.
Warning:
For security reasons, make sure that the Mnesia tables are stored outside the document tree of the Web server. If it is placed in the directory which it protects, clients will be able to download the tables.

Note:
Only the dets and mnesia storage methods allow writing of dynamic user data to disk. plain is a read only method.

DIRECTIVE: “AuthUserFile”

Syntax: AuthUserFile filename  
Default: -None-  
Module: mod_auth(3) [page 68]  
Context: <Directory> [page 68]  
Related: allow [page 72], deny [page 72], AuthDBType [page 69], AuthAccessPassword [page 72], AuthGroupFile [page 71], AuthName [page 71], require [page 73]  

AuthUserFile sets the name of a file which contains the list of users and passwords for user authentication. filename can be either absolute or relative to the ServerRoot.

If using the plain storage method, this file is a plain text file, where each line contains a user name followed by a colon, followed by the non-encrypted password. The behavior is undefined if user names are duplicated. For example:

ragnar:s7Xxv7  
edward:wwjau8

If using the dets storage method, the user database is maintained by dets and should not be edited by hand. Use the API [page 74] in this module to create / edit the user database.

This directive is ignored if using the mnesia storage method.

Warning:
For security reasons, make sure that the AuthUserFile is stored outside the document tree of the Web server. If it is placed in the directory which it protects, clients will be able to download it.
DIRECTIVE: ”AuthGroupFile”

Syntax: AuthGroupFile filename
Default: None
Module: mod_auth(3) [page 68]
Context: <Directory> [page 68]
Related: allow [page 72], deny [page 72], AuthName [page 71], AuthUserFile [page 70], AuthDBType [page 69], AuthAccessPassword [page 72], require [page 73]

AuthGroupFile sets the name of a file which contains the list of user groups for user authentication. filename can be either absolute or relative to the ServerRoot.

If you use the plain storage method, the group file is a plain text file, where each line contains a group name followed by a colon, followed by the member user names separated by spaces. For example:

group1: bob joe ante

If using the dets storage method, the group database is maintained by dets and should not be edited by hand. Use the API [page 74] in this module to create / edit the group database.

This directive is ignored if using the mnesia storage method.

Warning:
For security reasons, make sure that the AuthGroupFile is stored outside the document tree of the Web server. If it is placed in the directory which it protects, clients will be able to download it.

DIRECTIVE: ”AuthName”

Syntax: AuthName auth-domain
Default: None
Module: mod_auth(3) [page 68]
Context: <Directory> [page 68]
Related: allow [page 72], deny [page 72], AuthGroupFile [page 71], AuthUserFile [page 70], AuthDBType [page 69], AuthAccessPassword [page 72], require [page 73]

AuthName sets the name of the authorization realm (auth-domain) for a directory. This string informs the client about which user name and password to use.
DIRECTIVE: "AuthAccessPassword"

Syntax: AuthAccessPassword password
Default: NoPassword
Module: mod_auth(3) [page 68]
Context: <Directory> [page 68]
Related: allow [page 72], deny [page 72], AuthGroupFile [page 71], AuthUserFile [page 70], AuthDBType [page 69], AuthName [page 71], require [page 73]

If AuthAccessPassword is set to other than NoPassword the password is required for all API calls. If the password is set to DummyPassword the password must be changed before any other API calls. To secure the authenticating data the password must be changed after the webserver is started since it otherwise is written in clear text in the configuration file.

DIRECTIVE: "allow"

Syntax: allow from host host ...
Default: allow from all
Module: mod_auth(3) [page 68]
Context: <Directory> [page 68]
Related: AuthAccessPassword [page 72], deny [page 72], AuthUserFile [page 70], AuthGroupFile [page 71], AuthName [page 71], AuthDBType [page 69], require [page 73]

allow defines a set of hosts which should be granted access to a given directory. host is one of the following:

all All hosts are allowed access.

A regular expression (Read regexp(3)) All hosts having a numerical IP address matching the specific regular expression are allowed access.

For example:
allow from 123.34.56.11 150.100.23

The host 123.34.56.11 and all machines on the 150.100.23 subnet are allowed access.

DIRECTIVE: "deny"

Syntax: deny from host host ...
Default: deny from all
Module: mod_auth(3) [page 68]
Context: <Directory> [page 68]
Related: allow [page 72], AuthUserFile [page 70], AuthGroupFile [page 71], AuthName [page 71], AuthDBType [page 69], AuthAccessPassword [page 72], require [page 73]

deny defines a set of hosts which should not be granted access to a given directory. host is one of the following:

all All hosts are denied access.
A regular expression (Read regexp(3)) All hosts having a numerical IP address matching the specific regular expression are denied access.

For example:

deny from 123.34.56.11 150.100.23

The host 123.34.56.11 and all machines on the 150.100.23 subnet are denied access.

DIRECTIVE: "require"

Syntax: require entity-name entity entity ...
Default: - None -
Module: mod_auth(3) [page 68]
Context: <Directory> [page 68]
Related: allow [page 72], deny [page 72], AuthUserFile [page 70], AuthGroupFile [page 71], AuthName [page 71], AuthDBType [page 69], AuthAccessPassword [page 72]

require defines users which should be granted access to a given directory using a secret password. The allowed syntaxes are:

require user user-name user-name ...
Only the named users can access the directory.

require group group-name group-name ...
Only users in the named groups can access the directory.

EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:

{real_name, {Path, AfterPath}} as defined in mod_alias(3) [page 65].

Exports the following EWSAPI interaction data, if possible:

{remote_user, User} The user name with which the user has authenticated himself.

Uses the following exported EWSAPI functions:

- mod_alias:path/3 [page 67]
Exports

add_user(UserName, Options) -> true | {error, Reason}
add_user(UserName, Password, UserData, Port, Dir) -> true | {error, Reason}
add_user(UserName, Password, UserData, Address, Port, Dir) -> true | {error, Reason}

Types:
- UserName = string()
- Options = [Option]
  - Option = [password,Password] | [userData,UserData] | [port,Port] |
    [addr,Address] | [dir,Directory] | [authPassword,AuthPassword]
- Password = string()
- UserData = term()
- Port = integer()
- Address = [A,B,C,D] | string() | undefined
- Dir = string()
- AuthPassword = string()
- Reason = term()

add_user/2, add_user/5 and add_user/6 adds a user to the user database. If the
operation is successful, this function returns true. If an error occurs, {error,Reason} is
returned. When add_user/2 is called the Password, UserData Port and Dir options are
mandatory.

delete_user(UserName, Options) -> true | {error, Reason}
delete_user(UserName, Port, Dir) -> true | {error, Reason}
delete_user(UserName, Address, Port, Dir) -> true | {error, Reason}

Types:
- UserName = string()
- Options = [Option]
  - Option = [port,Port] | [addr,Address] | [dir,Directory] | [authPassword,AuthPassword]
- Port = integer()
- Address = [A,B,C,D] | string() | undefined
- Dir = string()
- AuthPassword = string()
- Reason = term()

delete_user/2, delete_user/3 and delete_user/4 deletes a user from the user
database. If the operation is successful, this function returns true. If an error occurs,
{error,Reason} is returned. When delete_user/2 is called the Port and Dir options are
mandatory.

get_user(UserName, Options) -> {ok, #httpd_user} | {error, Reason}
get_user(UserName, Port, Dir) -> {ok, #httpd_user} | {error, Reason}
get_user(UserName, Address, Port, Dir) -> {ok, #httpd_user} | {error, Reason}

Types:
- UserName = string()
Inets Reference Manual

- Options = [Option]
- Option = {port,Port} | {addr,Address} | {dir,Directory} | {authPassword,AuthPassword}
- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- Dir = string()
- AuthPassword = string()
- Reason = term()

get_user/2, get_user/3 and get_user/4 returns a httpd_user record containing the userdata for a specific user. If the user cannot be found, {error, Reason} is returned. When get_user/2 is called the Port and Dir options are mandatory.

list_users(Options) -> {ok, Users} | {error, Reason}
list_users(Port, Dir) -> {ok, Users} | {error, Reason}
list_users(Address, Port, Dir) -> {ok, Users} | {error, Reason}

Types:
- Options = [Option]
- Option = {port,Port} | {addr,Address} | {dir,Directory} | {authPassword,AuthPassword}
- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- Dir = string()
- Users = list()
- AuthPassword = string()
- Reason = atom()

list_users/1, list_users/2 and list_users/3 returns a list of users in the user database for a specific Port/Dir. When list_users/1 is called the Port and Dir options are mandatory.

add_group_member(GroupName, UserName, Options) -> true | {error, Reason}
add_group_member(GroupName, UserName, Port, Dir) -> true | {error, Reason}
add_group_member(GroupName, UserName, Address, Port, Dir) -> true | {error, Reason}

Types:
- GroupName = string()
- UserName = string()
- Options = [Option]
- Option = {port,Port} | {addr,Address} | {dir,Directory} | {authPassword,AuthPassword}
- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- Dir = string()
- AuthPassword = string()
- Reason = term()
add_group_member/3, add_group_member/4 and add_group_member/5 adds a user to a group. If the group does not exist, it is created and the user is added to the group. Upon successful operation, this function returns true. When add_group_members/3 is called the Port and Dir options are mandatory.

delete_group_member(GroupName, UserName, Options) -> true | {error, Reason}
deelete_group_member(GroupName, UserName, Port, Dir) -> true | {error, Reason}
delete_group_member(GroupName, UserName, Address, Port, Dir) -> true | {error, Reason}

Types:
- GroupName = string()
- UserName = string()
- Options = [Option]
- Option = {port, Port} | {addr, Address} | {dir, Directory} | {authPassword, AuthPassword}
- Port = integer()
- Address = {A, B, C, D} | string() | undefined
- Dir = string()
- AuthPassword = string()
- Reason = term()

delete_group_member/3, delete_group_member/4 and delete_group_member/5 deletes a user from a group. If the group or the user does not exist, this function returns an error, otherwise it returns true. When delete_group_member/3 is called the Port and Dir options are mandatory.

list_group_members(GroupName, Options) -> {ok, Users} | {error, Reason}
list_group_members(GroupName, Port, Dir) -> {ok, Users} | {error, Reason}
list_group_members(GroupName, Address, Port, Dir) -> {ok, Users} | {error, Reason}

Types:
- GroupName = string()
- Options = [Option]
- Option = {port, Port} | {addr, Address} | {dir, Directory} | {authPassword, AuthPassword}
- Port = integer()
- Address = {A, B, C, D} | string() | undefined
- Dir = string()
- Users = list()
- AuthPassword = string()
- Reason = term()

list_group_members/2, list_group_members/3 and list_group_members/4 lists the members of a specified group. If the group does not exist or there is an error, {error, Reason} is returned. When list_group_members/2 is called the Port and Dir options are mandatory.

list_groups(Options) -> {ok, Groups} | {error, Reason}
list_groups(Port, Dir) -> {ok, Groups} | {error, Reason}
list_groups(Address, Port, Dir) -> {ok, Groups} | {error, Reason}

Types:
- Options = [Option]
- Option = {port, Port} | {addr, Address} | {dir, Directory} | {authPassword, AuthPassword}
- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- Dir = string()
- Groups = list()
- AuthPassword = string()
- Reason = term()

list_groups/1, list_groups/2 and list_groups/3 lists all the groups available. If there is an error, {error, Reason} is returned. When list_groups/1 is called the Port and Dir options are mandatory.

delete_group(GroupName, Options) -> true | {error, Reason}
delete_group(GroupName, Port, Dir) -> true | {error, Reason}
delete_group(GroupName, Address, Port, Dir) -> true | {error, Reason}

Types:
- Options = [Option]
- Option = {port, Port} | {addr, Address} | {dir, Directory} | {authPassword, AuthPassword}
- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- Dir = string()
- GroupName = string()
- AuthPassword = string()
- Reason = term()

delete_group/2, delete_group/3 and delete_group/4 deletes the group specified and returns true. If there is an error, {error, Reason} is returned. When delete_group/2 is called the Port and Dir options are mandatory.

update_password(Port, Dir, OldPassword, NewPassword, NewPassword) -> ok | {error, Reason}
update_password(Address, Port, Dir, OldPassword, NewPassword, NewPassword) -> ok | {error, Reason}

Types:
- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- Dir = string()
- GroupName = string()
- OldPassword = string()
- NewPassword = string()
- Reason = term()
update_password/5 and update_password/6 Updates the AuthAccessPassword for the specified directory. If NewPassword is equal to "NoPassword" no password is required to change authorisation data. If NewPassword is equal to "DummyPassword" no changes can be done without changing the password first.

SEE ALSO

httpd(3) [page 40], mod_alias(3) [page 65],
mod_browser

Erlang Module

When a client requests for an asset the request-header may contain a string that identifies the product. Many browsers also sends information about which operating-system the client use. This can be used in conjunction with mod esi to tailor the response according to the users operating-system and browser.

This module can be used to recognize the browser and operating-system of the client in two ways either as a module in the EWSAPI response chain or by a separate call to the function getBrowser/1.

Exports

getBrowser(AgentString) -> {Browser,OperatingSystem}

Types:

- AgentString = string()
- Browser = {Name,Version} | unknown
- OperatingSystem = win3x | win95 | win98 | winnt | win2k | sunos4 | sunos5 | sun | aix | linux | sco | freebsd | bsd | macosx | unknown
- Name = opera | msie | netscape | lynx | mozilla | emacs | soffice | mosaic | safari
- Version = float().

The function getBrowser/1, tries to detect which browser and operating system the user has. Note that the answer is just a best guess since some browsers can identify themselves as other browsers, read Opera.

EWSAPI MODULE INTERACTION

Exports the following EWSAPI interaction data, if possible:

{‘user-agent’, AgentData} Where AgentData is the same as the return value from getBrowser/1. Note that the answer is just a best guess, since some browsers can identify themselves as other browsers, read Opera.
mod_cgi

Erlang Module

This module makes it possible to execute vanilla CGI (Common Gateway Interface) scripts in the server. A file that matches the definition of a ScriptAlias [page 66] config directive is treated as a CGI script. A CGI script is executed by the server and it's output is returned to the client.

The CGI Script response comprises a message-header and a message-body, separated by a blank line. The message-header contains one or more header fields. The body may be empty. Example:

"Content-Type:text/plain
Accept-Ranges:none

some very plain text"

The server will interpret the cgi-headers and most of them will be transformed into HTTP headers and sent back to the client.

Support for CGI-1.1 is implemented in accordance with the RFC 3875.8

- ScriptNoCache [page 80]
- ScriptTimeout [page 80]

DIRECTIVE: "ScriptNoCache"

Syntax: ScriptNoCache true | false
Default: false
Module: mod_cgi(3) [page 80]
If ScriptNoCache is set to true the Web server will by default add the header fields necessary to prevent proxies from caching the page. Generally this is something you want.

ScriptNoCache true

DIRECTIVE: "ScriptTimeout"

Syntax: ScriptTimeout Seconds
Default: 15
Module: mod_cgi(3) [page 80]
The time in seconds the web server will wait between each chunk of data from the script. If the CGI-script not delivers any data before the timeout the connection to the client will be closed.

ScriptTimeout 15

8URL: http://www.faqs.org/rfcs/rfc3875.html
EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:

\{new_request_uri, NewRequestURI\} as defined in mod_actions(3) [page 64].
\{remote_user, RemoteUser\} as defined in mod_auth(3) [page 73].

Uses the following EWSAPI functions:

- mod_alias:real_name/3 [page 67]
- mod_alias:real_script_name/3 [page 67]
- mod cgi:env/3 [page 81]
- mod cgi:status_code:env/1 [page 81]

This module furthermore exports a batch of functions to be used by other EWSAPI modules:

Exports

\text{env(Info, Script, AfterScript) -> EnvString}

Types:
\begin{itemize}
  \item Info = mod_record()
  \item Script = AfterScript = EnvString = string()
\end{itemize}

\text{Note:}
This function should only be used when implementing CGI-1.1 functionality on UNIX platforms.

\text{open_port/2} is normally used to start and interact with CGI scripts. \text{open_port/2} takes an external program as input; \text{env(1)} (GNU Shell Utility) is typically used in the case of a CGI script. \text{env(1)} execute the CGI script in a modified environment and takes the CGI script and a string of environment variables as input. \text{env/3} returns an appropriate CGI-1.1 environment variable string to be used for this purpose. The environment variables in the string are those defined in the RFC 3875\textsuperscript{9} mod_record() is a record as defined in the EWSAPI Module Programming in the Inets Users Guide [page 4].

\text{status_code(CGIOutput) -> {ok, StatusCode} | {error, Reason}}

Types:
\begin{itemize}
  \item CGIOutput = Reason = string()
  \item StatusCode = integer()
\end{itemize}

Certain output from CGI scripts has a special meaning, as described in the RFC 3875\textsuperscript{10}, for example if "Location: http://www.yahoo.com\n\n" is returned from a CGI script the client gets automatically redirected to Yahoo!\textsuperscript{11}, using the HTTP 302 status code.

\textsuperscript{9}\text{URL: http://www.faqs.org/rfcs/rfc3875.html}
\textsuperscript{10}\text{URL: http://www.faqs.org/rfcs/rfc3875.html}
\textsuperscript{11}\text{URL: http://www.yahoo.com}
SEE ALSO

httpd(3) [page 40], mod_auth(3) [page 68], mod_security(3) [page 109], mod_alias(3) [page 65], mod_esi(3) [page 88], mod_include(3) [page 101]
mod_dir
Erlang Module

This module generates an HTML directory listing (Apache-style) if a client sends a request for a directory instead of a file. This module is not configurable and it needs to be removed from the Modules [page 51] config directive if directory listings is unwanted.

EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:

\{real_name,\{Path,AfterPath\}\} as defined in mod_alias(3) [page 66].

Exports the following EWSAPI interaction data, if possible:

\{mime_type,MimeType\} The file suffix of the incoming URL mapped into a MimeType as defined in the Mime Type Settings [page 48] section of httpd_core(3).

Uses the following EWSAPI functions:

- mod_alias:default_index/2 [page 66]
- mod_alias:path/3 [page 67]

SEE ALSO

httpd(3) [page 40], mod_alias(3) [page 65]
mod_disk_log

Erlang Module

This module uses `disk_log(3)` to make it possible to log all incoming requests to an access log file. The de-facto standard Common Logfile Format is used for this purpose. There are numerous statistic programs available to analyze Common Logfile Format log files. The Common Logfile Format looks as follows:

remotehost rfc931 authuser [date] “request” status bytes

- **remotehost**  Remote hostname (or IP number if the DNS hostname is not available).
- **rfc931**  The client’s remote username (RFC 931).
- **authuser**  The username with which the user has authenticated himself.
- **[date]**  Date and time of the request (RFC 1123).
- **“request”**  The request line exactly as it came from the client (RFC 1945).
- **status**  The HTTP status code returned to the client (RFC 1945).
- **bytes**  The content-length of the document transferred.

This module furthermore uses `disk_log(3)` to support the use of an error log file to record internal server errors. The error log format is more ad hoc than Common Logfile Format, but conforms to the following syntax:

[date] access to path failed for remotehost, reason: reason

**DIRECTIVE: “DiskLogFormat”**

Syntax: DiskLogFormat internal | external
Default: - external -
Module: `mod_disk_log(3)` [page 84]

`DiskLogFormat` defines the file-format of the log files see `disk_log` for more information. If the internal file-format is used, the logfile will be repaired after a crash. When a log file is repaired data might get lost. When the external file-format is used `httpd` will not start if the log file is broken.

```
DiskLogFormat external
```
DIRECTIVE: "ErrorDiskLog"

Syntax: ErrorDiskLog filename
Default: - None -
Module: mod_disk_log(3) [page 84]

ErrorDiskLog defines the filename of the (disk_log(3)) error log file to be used to log server errors. If the filename does not begin with a slash (/) it is assumed to be relative to the ServerRoot [page 52], for example:

ErrorDiskLog logs/error_disk_log_8080

and errors will be logged in the server root space.

DIRECTIVE: "ErrorDiskLogSize"

Syntax: ErrorDiskLogSize max-bytes max-files
Default: ErrorDiskLogSize 512000 8
Module: mod_disk_log(3) [page 84]

ErrorDiskLogSize defines the properties of the (disk_log(3)) error log file. The disk_log(3) error log file is of type wrap log and max-bytes will be written to each file and max-files will be used before the first file is truncated and reused.

DIRECTIVE: "SecurityDiskLog"

Syntax: SecurityDiskLog filename
Default: - None -
Module: mod_disk_log(3) [page 84]

SecurityDiskLog defines the filename of the (disk_log(3)) access log file which logs incoming security events, i.e authenticated requests. If the filename does not begin with a slash (/) it is assumed to be relative to the ServerRoot [page 52], see TransferDiskLog [page 86] for more information.

DIRECTIVE: "SecurityDiskLogSize"

Syntax: SecurityDiskLogSize max-bytes max-files
Default: SecurityDiskLogSize 512000 8
Module: mod_disk_log(3) [page 84]

SecurityDiskLogSize defines the properties of the disk_log(3) access log file. The disk_log(3) access log file is of type wrap log and max-bytes will be written to each file and max-files will be used before the first file is truncated and reused.

DIRECTIVE: "TransferDiskLog"

Syntax: TransferDiskLog filename
Default: - None -
Module: mod_disk_log(3) [page 84]
TransferDiskLog defines the filename of the (disk_log(3)) access log file which logs incoming requests. If the filename does not begin with a slash (/) it is assumed to be relative to the ServerRoot [page 52], for example:

TransferDiskLog logs/transfer_disk_log_8080

and errors will be logged in the server root\textsuperscript{13} space.

DIRECTIVE: "TransferDiskLogSize"

Syntax: TransferDiskLogSize max-bytes max-files
Default: TransferDiskLogSize 512000 8
Module: mod_disk_log(3) [page 84]
TransferDiskLogSize defines the properties of the disk_log(3) access log file. The disk_log(3) access log file is of type wrap log and max-bytes will be written to each file and max-files will be used before the first file is truncated and reused.

EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:

{remote_user,RemoteUser} as defined in mod_auth(3) [page 73].

This module furthermore exports a batch of functions to be used by other EWSAPI modules:

Exports

error_log(Socket,SocketType,ConfigDB,Date,Reason) -> ok | no_error_log

Types:
- Socket = socket()
- SocketType = ip_comm | ssl
- ConfigDB = config_db()
- Date = Reason = string()

error_log/5 uses disk_log(3) to log an error in the error log file. Socket is a handler to a socket of type SocketType and config_db() is the server config file in ETS table format as described in httpd(3) [page 40]. Date is a RFC 1123 date string as generated by httpd_util:rfc1123_date/0 [page 61].

\textsuperscript{13}In Windows: %SERVER_ROOT%\logs\transfer_disk_log_8080. In UNIX: $SERVER_ROOT/logs/transfer_disk_log_8080.
\texttt{security\_log}(\texttt{User},\texttt{Event}) \rightarrow \texttt{ok} \mid \texttt{no\_security\_log}

Types:
- \texttt{User} = \texttt{String()}
- \texttt{Event} = \texttt{String}

\texttt{security\_log/2 uses disk\_log(3)} to log a security event in the security log file. \texttt{User} is the users name.

\textbf{SEE ALSO}

\texttt{httpd(3) [page 40], mod\_auth(3) [page 68], mod\_security(3) [page 109], mod\_log(3) [page 104]}
The Erlang Scripting Interface (ESI) provides a tight and efficient interface to the execution of Erlang functions. Erlang functions can be executed with two alternative schemes, eval and erl. Both of these schemes can utilize the functionality in an Erlang node efficiently.

Even though the server supports CGI-1.1 [page 80] the use of the Erlang Scripting Interface (ESI) is encouraged for reasons of efficiency. CGI is resource intensive because of its design. CGI requires the server to fork a new OS process for each executable it needs to start.

An Erlang function can be written and executed as a CGI script by using erl:call(3) in the erl_interface library, for example. The cost is a forked OS process, as described above. This is a waste of resources, at least when the Web server itself is written in Erlang (as in this case).

The following config directives are described:

- ErlScriptAlias [page 91]
- EvalScriptAlias [page 92]
- ErlScriptNoCache [page 91]
- ErlScriptTimeout [page 92]

**ERL SCHEME**

The erl scheme is designed to mimic plain CGI, but without the extra overhead. An URL which calls an Erlang erl function has the following syntax (regular expression): http://your.server.org/***/Mod[:/]Func(?QueryString|/PathInfo)

The module (Mod) referred to must be found in the code path, and it must define a function (Func) with an arity of two or three i.e. `Func(Env,Input)` or `Func(SessionID,Env,Input)`. Env contains information about the connecting client (see below), and Input the QueryString or PathInfo as defined in the CGI specification14. SessionID is a identifier that is used to send parts of the web page back to the user through the function mod_esi:deliver/2

*** above depends on how the ErlScriptAlias [page 91] config directive has been used. Data returned from the function with arity of two must furthermore take the form as specified in the CGI specification15.

It is preferable to use the callback function with an arity of three, since the function can send the data back to the clients in parts instead of generating the whole page before it

---

14 URL: http://hoohoo.ncsa.uiuc.edu/cgi/
15 URL: http://hoohoo.ncsa.uiuc.edu/cgi/
is sent. The Web server sends the data back to HTTP/1.1 compliant clients with chunked encoding this means that the Content-Length header field is not necessary, and should indeed be avoided.

mod_esi assumes that if the first chunk of data delivered to the client through the function mod_esi:deliver/2 contains all HTTP-header fields the script will generate. If the first chunk does not contain the string "\r\n\r\n"c mod_esi assumes that the script not will generate any header data.

Take a look at httpd_example.erl in the code release\(^\text{16}\) for a clarifying example. Start an example server as described in httpd:start/0 [page 40] and test the following from a browser (The server name for your example server will differ!):

http://your.server.org:8888/cgi-bin/erl/httpd_example/newformat and a call will be made to httpd_example:newformat/3 Something like this will promptly be shown in the browser:

This new format is nice.
This new format is nice.
This new format is nice.

http://your.server.org:8888/cgi-bin/erl/httpd_example/get and a call will be made to httpd_example:get/2 and two input fields and a Submit button will promptly be shown in the browser. Enter text into the input fields and click on the Submit button. Something like this will promptly be shown in the browser:

Environment:

```
{{query_string,"input1=blaha&input2=blaha"},
{server_software,"eddie/2.2"},
{server_name,"localhost"},
{gateway_interface,"CGI/1.1"},
{server_protocol,"HTTP/1.0"},
{server_port,8080},
{request_method,"GET"},
{remote_addr,"127.0.0.1"},
{script_name, "/cgi-bin/erl/httpd_example:get?input1=blaha&input2=blaha"},
{http_accept_charset,"iso-8859-1,*,utf-8"},
{http_accept_language,"en"},
{http_accept,"image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*"},
{http_host,"localhost:8080"},
{http_user_agent, "Mozilla/4.03 [en] (X11; I; Linux 2.0.30 i586)"},
{http_connection,"Keep-Alive"},
{http_referer, "http://localhost:8080/cgi-bin/erl/httpd_example/get"}]
```

Input:

```
input1=blaha&input2=blaha
```

Parsed Input:

\(^\text{16}\)In Windows: %INETS\src. In UNIX: $INETS/src.
{{"input1","blaha"},{"input2","blaha"}}

http://your.server.org:8888/cgi-bin/erl/httpd_example:post  A call will be made to httpd_example:post/2. The same thing will happen as in the example above but the HTTP POST method will be used instead of the HTTP GET method.

http://your.server.org:8888/cgi-bin/erl/httpd_example:yahoo  A call will be made to httpd_example:yahoo/2 and the Yahoo!17 site will promptly be shown in your browser.

**Note:**

httpd:parse_query/1 [page 42] is used to generate the Parsed Input: ... data in the example above.

If a client closes the connection prematurely a message will be sent to the function, that is either {tcp,closed,1} or {socket,closed,1}.

### EVAL SCHEME

**Warning:**
The eval scheme can seriously threaten the integrity of the Erlang node housing a Web server, for example:

```
http://your.server.org/eval?httpd_example:
  print(atom_to_list(apply(erlang,halt,[])))
```

which effectively will close down the Erlang node, that is use the erl scheme instead until this security breach has been fixed.

Today there are no good way of solving this problem and therefore Eval Scheme may be removed in future releases of Inets.

The eval scheme is straight-forward and does not mimic the behavior of plain CGI. An URL which calls an Erlang eval function has the following syntax:

```
http://your.server.org/***/Mod:Func(Arg1,...,ArgN)
```

The module (Mod) referred to must be found in the code path, and data returned by the function (Func) is passed back to the client. *** depends on how the EvalScriptAlias [page 92] config directive has been used. Data returned from the function must furthermore take the form as specified in the CGI specification18.

Take a look at httpd_example.erl in the code release19 for an example. Start an example server as described in httpd:start/0 [page 40] and test the following from a browser (The server name for your example server will differ!):

---

17URL: http://www.yahoo.com
18URL: http://hoohoo.ncsa.uiuc.edu/cgi/
19In Windows: %INETS\src. In UNIX: $INETS/src.
http://your.server.org:8888/eval?httpd_example:print("Hi!") and a call will be made to httpd_example:print/1 and “Hi!” will promptly be shown in your browser.

**DIRECTIVE: “ErlScriptAlias”**

Syntax: ErlScriptAlias url-path allowed-module allowed-module ...
Default: - None -
Module: mod_esi(3) [page 88]

ErlScriptAlias marks all URLs matching url-path as Erl scheme [page 88] scripts. A matching URL is mapped into a specific module and function. The module must be one of the allowed-module:s. For example:

ErlScriptAlias /cgi-bin/hit_me httpd_example md4

and a request to http://your.server.org/cgi-bin/hit_me/httpd_example:yahoo would refer to httpd_example:yahoo/2. Refer to the Erl Scheme [page 88] description above.

**DIRECTIVE: “ErlScriptNoCache”**

Syntax: ErlScriptNoCache true | false
Default: false
Module: mod_esi(3) [page 88]

If ErlScriptNoCache is set to true the server will add http header fields that prevents proxies from caching the page. This is generally a good idea for dynamic content, since the content often vary between each request.

ErlScriptNoCache true

**DIRECTIVE: “ErlScriptTimeout”**

Syntax: ErlScriptTimeout seconds
Default: 15
Module: mod_esi(3) [page 88]

If ErlScriptTimeout sets the time in seconds the server will wait between each chunk of data is delivered through mod_esi:deliver/2 when the new Erl Scheme format, that takes three argument is used.

ErlScriptTimeout 15
DIRECTIVE: "EvalScriptAlias"

Syntax: EvalScriptAlias url-path allowed-module allowed-module ...
Default: - None-
Module: mod_esi(3) [page 88]

EvalScriptAlias marks all URLs matching url-path as eval scheme [page 90] scripts. A matching URL is mapped into a specific module and function. The module must be one of the allowed-modules. For example:

EvalScriptAlias /cgi-bin/hit me to httpd_example md5

and a request to http://your.server.org/cgi-bin/hit me to httpd_example:print("Hi!") would refer to httpd_example:print/1. Refer to the Eval Scheme [page 90] description above.

EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:

{remote_user,RemoteUser} as defined in mod_auth(3) [page 73].

Exports the following EWSAPI interaction data, if possible:

{mime_type,MimeType} The file suffix of the incoming URL mapped into a MimeType as defined in the Mime Type Settings [page 48] section of httpd_core(3).

Uses the following EWSAPI functions:

- mod_alias:real_name/3 [page 67]
- mod_cgi:status_code/1 [page 81]

Exports

deliver(SessionID, Data) -> ok | {error,Reason}

Types:
- SessionID = term()
- Data = string()
- Reason = term()

This function is only intended to be used from functions called by the Erl Scheme interface to deliver parts of the content to the user.

Sends data from a Erl Scheme script back to the client. Note that if any HTTP-header fields will be sent back to the client they must be in the first call to deliver/2. Do not assume anything about the data type of SessionID, the SessionID must be the SessionID from the Erl Scheme call.
ESWAPI CALLBACK FUNCTIONS

Exports

Module:Function(Env, Input) -> Response

Types:
- Env = [EnvironmentDirectives] ++ ParsedHeader
- EnvironmentDirectives = {[Key, Value]}
- Key = query_string | content_length, server_software, gateway_interface, server_protocol, server_port, request_method, remote_addr, script_name.
- Input = Response = string()

The Module must be found in the code path and export Function with an arity of two. An erlScriptAlias must also be set up in the configuration file for the Web server.

If the HTTP request is a post request and a body is sent, then content_length will be the length of the posted data. If get is used, query string will be the data after ? in the url.

ParsedHeader is the HTTP request as a key value tuple list. The keys in parsed header will be in lower case.

This callback format consumes quite much memory since the whole response must be generated before it is sent to the user. Therefore it is better to use the callback function with an arity of three.

Module:Function(SessionID, Env, Input) -> void

Types:
- SessionID = term()
- Env = [EnvironmentDirectives] ++ ParsedHeader
- EnvironmentDirectives = {[Key, Value]}
- Key = query_string | content_length, server_software, gateway_interface, server_protocol, server_port, request_method, remote_addr, script_name.
- Input = Response = string()

For information about Environment and Input see Module:Function/2 above.

SessionID is a identifier the server use when deliver/2 is called, do not assume any-thing about the datatype.

Use this callback function to dynamically generate dynamic web content. When a part of the page is generated send the data back to the client through deliver/2. Note that the first chunk of data sent to the client must at least contain all HTTP header fields that the response will generate. If the first chunk not contains End of HTTP header that is "\r\n\r\n" the server will assume that no HTTP header fields will be generated.

SEE ALSO

httpd(3) [page 40], mod_alias(3) [page 65], mod_auth(3) [page 68], mod_security(3) [page 109], mod_cgi(3) [page 80]
mod_get

Erlang Module

This module is responsible for handling GET requests to regular files. GET requests for parts of files is handled by mod_range.

EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:

{real_name, {Path, AfterPath}} as defined in mod_alias(3) [page 66].

Exports the following EWSAPI interaction data, if possible:

Uses the following EWSAPI functions:

- mod_alias:path/3 [page 67]

SEE ALSO

httpd(3) [page 40], mod_range(3) [page 107]
mod_head

Erlang Module

This module is responsible for handling HEAD requests to regular files. HEAD requests for dynamic content is handled by each module responsible for dynamic content.

EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:

{real_name, {Path, AfterPath}} as defined in mod_alias(3) [page 66].

Exports the following EWSAPI interaction data, if possible:

Uses the following EWSAPI functions:

- mod_alias:path/3 [page 67]

SEE ALSO

httpd(3) [page 40], mod_vhost(3) [page 88], mod_cgi(3) [page 88]
This module provides per-directory runtime configurable user-authentication. Each directory in the path to the requested asset is searched for an access-file (default .htaccess), that restricts the webserver's rights to respond to a request. If an access-file is found the rules in that file is applied to the request.

The rules in an access-file applies both to files in the same directories and in subdirectories. If there exists more than one access-file in the path to an asset, the rules in the access-file nearest the requested asset will be applied.

If many users have web pages on the webserver and every user needs to manage the security issues alone, use this module.

To change the rules that restricts the use of an asset. The user only needs to have write access to the directory where the asset exists.

When a request comes, the path to the requested asset is searched for access-files with the name specified by the AccessFileName parameter, default .htaccess. When such a file is found it is parsed and the restrictions in the file is applied to the request. This means that a user do not need to have access to the webserver's configuration-file to limit the access to an asset. Furthermore the user can change the rules and the changes will be applied immediately.

All the access-files in the path to a requested asset is read once per request, this means that the load on the server will increase when this module is used.

The following configuration directives are supported

- AccessFileName [page 96]

**DIRECTIVE: "AccessFileName"**

Syntax: AccessFileName FileName1 FileName2
Default: .htaccess Module: mod_htaccess(3) [page 96]

AccessFileName Specify which filenames that are used for access-files. When a request comes every directory in the path to the requested asset will be searched after files with the names specified by this parameter. If such a file is found the file will be parsed and the restrictions specified in it will be applied to the request.
Access Files Directives

In every directory under the DocumentRoot or under an Alias a user can place an access-file. An access-file is a plain text file that specify the restrictions that shall be considered before the webserver answer to a request. If there are more than one access-file in the path to the requested asset, the directives in the access-file in the directory nearest the asset will be used.

- allow [page 97]
- AllowOverRide [page 97]
- AuthGroupFile [page 98]
- AuthName [page 98]
- AuthType [page 98]
- AuthUserFile [page 98]
- deny [page 99]
- <Limit> [page 99]
- order [page 99]
- require [page 100]

**DIRECTIVE: ”allow”**

Syntax: Allow from subnet subnet | from all
Default: from all
Module: mod_\_htaccess(3) [page 96]
Context: <Limit> [page 99]
Related: mod_\_auth(3), [page 68]

See the allow directive in the documentation of mod_\_auth(3) for more information.

**DIRECTIVE: ”AllowOverRide”**

Syntax: AllowOverRide all | none | Directives
Default: - None -
Module: mod_\_htaccess(3) [page 96]

AllowOverRide Specify which parameters that not access-files in subdirectories are allowed to alter the value for. If the parameter is set to none no more access-files will be parsed.

If only one access-file exists setting this parameter to none can lessen the burden on the server since the server will stop looking for access-files.
DIRECTIVE: "AuthGroupFile"

Syntax: AuthGroupFile Filename
Default: - None -
Module: mod_htaccess(3) [page 96]
Related: mod_auth(3) [page 68],

AuthGroupFile indicates which file that contains the list of groups. Filename must
tcont the absolute path to the file. The format of the file is one group per row and
every row contains the name of the group and the members of the group separated by a
space, for example:

```
GroupName: Member1 Member2 .... MemberN
```

DIRECTIVE: "AuthName"

Syntax: AuthName auth-domain
Default: - None -
Module: mod_htaccess(3) [page 96]
Related: mod_auth(3) [page 68],

See the AuthName directive in the documentation of mod_auth(3) for more
information.

DIRECTIVE: "AuthType"

Syntax: AuthType Basic
Default: Basic
Module: mod_htaccess(3) [page 96]

AuthType Specify which authentication scheme that shall be used. Today only Basic
Authenticating using UUEncoding of the password and user ID is implemented.

DIRECTIVE: "AuthUserFile"

Syntax: AuthUserFile Filename
Default: - None -
Module: mod_htaccess(3) [page 96]
Related: mod_auth(3) [page 68],

AuthUserFile indicate which file that contains the list of users. Filename must contain
the absolute path to the file. The users name and password are not encrypted do not
place the file with users in a directory that is accessible via the webserver. The format of
the file is one user per row and every row contains User Name and Password separated
by a colon, for example:

```
UserName:Password
UserName:Password
```
DIRECTIVE: "deny"

Syntax: deny from subnet subnet | from all
Default: from all
Module: mod\_htaccess(3) [page 96]
Context: <Limit> [page 99]
Related: mod\_auth(3) [page 68],
See the deny directive in the documentation of mod\_auth(3) for more information.

DIRECTIVE: "Limit"

Syntax: <Limit RequestMethods>
Default: - None -
Module: mod\_auth(3) [page 68]
Related: order [page 99], allow [page 97], deny [page 99], require [page 100]

<Limit> and </Limit> are used to enclose a group of directives which applies only to requests using the specified methods. If no request method is specified all request methods are verified against the restrictions.

    <Limit POST GET HEAD>
    order allow deny
    require group group1
    allow from 123.145.244.5
    </Limit>

DIRECTIVE: "order"

Syntax: order allow deny | deny allow
Default: allow deny
Module: mod\_htaccess(3) [page 96]
Context: order [page 99]
Related: allow [page 97], deny [page 99]

order, defines if the deny or allow control shall be preformed first.

If the order is set to allow deny, then first the users network address is controlled to be in the allow subset. If the users network address is not in the allowed subset he will be denied to get the asset. If the network-address is in the allowed subset then a second control will be preformed, that the users network address is not in the subset of network addresses that shall be denied as specified by the deny parameter.

If the order is set to deny allow then only users from networks specified to be in the allowed subset will succeed to request assets in the limited area.
DIRECTIVE: "require"

Syntax: require group group1 group2... | user user1 user2...
Default: - None -
Context: <Limit> [page 99]
Module: mod_htaccess(3) [page 96]
Related: mod_auth(3) [page 68],
See the require directive in the documentation of mod_auth(3) for more information.

EWSAPI MODULE INTERACTION

If a directory is limited both by mod_auth and mod_htaccess the user must be allowed
to request the asset for both of the modules.
Uses the following EWSAPI interaction data, if available:

{real_name, {Path, AfterPath}} as defined in mod_alias(3) [page 65].

Exports the following EWSAPI interaction data, if possible:

{remote_user_name, User} The user name with which the user has authenticated
himself.

Uses the following exported EWSAPI functions:

- mod_alias:path/3 [page 67]
mod_include

Erlang Module

This module makes it possible to expand “macros” embedded in HTML pages before they are delivered to the client, that is Server-Side Includes (SSI). To make this possible the server parses HTML pages on-the-fly and optionally includes the current date, the requested file’s last modification date or the size (or last modification date) of other files. In its more advanced form, it can include output from embedded CGI and /bin/sh scripts.

**Note:**
Having the server parse HTML pages is a double edged sword! It can be costly for a heavily loaded server to perform parsing of HTML pages while sending them. Furthermore, it can be considered a security risk to have average users executing commands in the name of the Erlang node user. Carefully consider these items before activating server-side includes.

**SERVER-SIDE INCLUDES (SSI) SETUP**

The server must be told which filename extensions to be used for the parsed files. These files, while very similar to HTML, are not HTML and are thus not treated the same. Internally, the server uses the magic MIME type text/x-server-parsed-html to identify parsed documents. It will then perform a format conversion to change these files into HTML for the client. Update the mime.types file, as described in the Mime Type Settings [page 48] section of httpd(3), to tell the server which extension to use for parsed files, for example:

```
text/x-server-parsed-html shtml shtm
```

This makes files ending with .shtml and .shtm into parsed files. Alternatively, if the performance hit is not a problem, all HTML pages can be marked as parsed:

```
text/x-server-parsed-html html htm
```
SERVER-SIDE INCLUDES (SSI) FORMAT

All server-side include directives to the server are formatted as SGML comments within the HTML page. This is in case the document should ever find itself in the client's hands unparsed. Each directive has the following format:

\[<!-#command tag1="value1" tag2="value2" -->\]

Each command takes different arguments, most only accept one tag at a time. Here is a breakdown of the commands and their associated tags:

**config** – The config directive controls various aspects of the file parsing. There are two valid tags:

- **errmsg** – controls the message sent back to the client if an error occurred while parsing the document. All errors are logged in the server's error log.
- **sizefmt** – determines the format used to display the size of a file. Valid choices are `bytes` for a formatted byte count or `abbrev` for an abbreviated version displaying the number of kilobytes.

**include** – will insert the text of a document into the parsed document. This command accepts two tags:

- **virtual** – gives a virtual path to a document on the server. Only normal files and other parsed documents can be accessed in this way.
- **file** – gives a pathname relative to the current directory. `../` cannot be used in this pathname, nor can absolute paths. As above, you can send other parsed documents, but you cannot send CGI scripts.

**echo** – prints the value of one of the include variables (defined below). The only valid tag to this command is `var`, whose value is the name of the variable you wish to echo.

**fsize** – prints the size of the specified file. Valid tags are the same as with the include command. The resulting format of this command is subject to the sizefmt parameter to the config command.

**flastmod** – prints the last modification date of the specified file. Valid tags are the same as with the include command.

**exec** – executes a given shell command or CGI script. Valid tags are:

- **cmd** – executes the given string using `/bin/sh`. All of the variables defined below are defined, and can be used in the command.
- **cgi** – executes the given virtual path to a CGI script and includes its output. The server does not perform error checking on the script output.

SERVER-SIDE INCLUDES (SSI) ENVIRONMENT VARIABLES

A number of variables are made available to parsed documents. In addition to the CGI variable set, the following variables are made available:

- **DOCUMENT_NAME** – The current filename.
- **DOCUMENT_URI** – The virtual path to this document (such as `/docs/tutorials/foo.shtml`).
QUERY\_STRING\_UNESCAPED  The unescaped version of any search query the client sent, with all shell-special characters escaped with \.

DATE\_LOCAL  The current date, local time zone.

DATE\_GMT  Same as DATE\_LOCAL but in Greenwich mean time.

LAST\_MODIFIED  The last modification date of the current document.

EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:

{real\_name, {Path, AfterPath}} as defined in mod\_alias(3) [page 66].
{remote\_user, RemoteUser} as defined in mod\_auth(3) [page 73]

Exports the following EWSAPI interaction data, if possible:

{mime\_type, MimeType} The file suffix of the incoming URL mapped into a MimeType as defined in the Mime Type Settings [page 48] section of httpd\_core(3).

Uses the following EWSAPI functions:

- mod\_cgi:env/3 [page 81]
- mod\_alias:path/3 [page 67]
- mod\_alias:real\_name/3 [page 67]
- mod\_alias:real\_script\_name/3 [page 67]

SEE ALSO

httpd(3) [page 40], mod\_alias(3) [page 65], mod\_auth(3) [page 68], mod\_security(3) [page 109], mod\_cgi(3) [page 80]
This module makes it possible to log all incoming requests to an access log file. The de-facto standard Common Logfile Format is used for this purpose. There are numerous statistics programs available to analyze Common Logfile Format. The Common Logfile Format looks as follows:

```
remotehost rfc931 authuser [date] “request” status bytes
```

- **remotehost**: Remote hostname
- **rfc931**: The client’s remote username (RFC 931).
- **authuser**: The username with which the user authenticated himself.
- **[date]**: Date and time of the request (RFC 1123).
- **“request”**: The request line exactly as it came from the client (RFC 1945).
- **status**: The HTTP status code returned to the client (RFC 1945).
- **bytes**: The content-length of the document transferred.

This module furthermore supports the use of an error log file to record internal server errors. The error log format is more ad hoc than Common Logfile Format, but conforms to the following syntax:

```
[date] access to path failed for remotehost, reason: reason
```

### DIRECTIVE: “ErrorLog”

Syntax: `ErrorLog` filename
Default: `-None-
Module: `mod_log`(3) [page 104]

`ErrorLog` defines the `filename` of the error log file to be used to log server errors. If the `filename` does not begin with a slash (/) it is assumed to be relative to the `ServerRoot` [page 52], for example:

```
ErrorLog logs/error_log_8080
```

and errors will be logged in the server root space.

---

20In Windows: `%SERVER_ROOT%\logs\error_log_8080`. In UNIX: `%SERVER_ROOT%/logs/error_log_8080`. 
DIRECTIVE: "SecurityLog"

Syntax: SecurityLog filename
Default: -None-
Module: mod_log(3) [page 104]

SecurityLog defines the filename of the access log file to be used to log security events. If the filename does not begin with a slash (/) it is assumed to be relative to the ServerRoot [page 52]. For example:

SecurityLog logs/security_log_8080

and security events will be logged in the server root\(^{21}\) space.

DIRECTIVE: "TransferLog"

Syntax: TransferLog filename
Default: -None-
Module: mod_log(3) [page 104]

TransferLog defines the filename of the access log file to be used to log incoming requests. If the filename does not begin with a slash (/) it is assumed to be relative to the ServerRoot [page 52]. For example:

TransferLog logs/access_log_8080

and errors will be logged in the server root\(^{22}\) space.

EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:

\{remote_user,RemoteUser\} as defined in mod_auth(3) [page 73].

This module furthermore exports a batch of functions to be used by other EWSAPI modules:

\(^{21}\)In Windows: %SERVER\_ROOT%\logs\security\_log\_8080. In UNIX: $SERVER\_ROOT/logs/security\_log\_8080.

\(^{22}\)In Windows: %SERVER\_ROOT%\logs\access\_log\_8080. In UNIX: $SERVER\_ROOT/logs/access\_log\_8080.
Exports

error_log(Socket,SocketType,ConfigDB,Date,Reason) -> ok | no_error_log

Types:
- Socket = socket()
- SocketType = ip_com | ssl
- ConfigDB = config_db()
- Date = Reason = string()

error_log/5 logs an error in a log file. Socket is a handler to a socket of type SocketType and config_db() is the server config file in ETS table format as described in httpd(3) [page 40]. Date is a RFC 1123 date string as generated by httpd_util:rfc1123_date/0 [page 61].

SEE ALSO

httpd(3) [page 40], mod_auth(3) [page 68], mod_security(3) [page 109], mod_disklog(3) [page 84]
mod_range

Erlang Module

This module response to requests for one or many ranges of a file. This is especially useful when downloading large files, since a broken download may be resumed. Note that request for multiple parts of a document will report a size of zero to the log file.

EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:
{real_name, [Path, AfterPath]} as defined in mod_alias(3) [page 66].

Uses the following EWSAPI functions:

* mod_alias:path/3 [page 67]

SEE ALSO

httpd(3) [page 40], mod_get(3) [page 65]
This module controls that the conditions in the requests is fulfiled. For example a
request may specify that the answer only is of interest if the content is unchanged since
last retrieval. Or if the content is changed the range-request shall be converted to a
request for the whole file instead.

If a client sends more then one of the header fields that restricts the servers right to
respond, the standard does not specify how this shall be handled. httpd will control
each field in the following order and if one of the fields not match the current state the
request will be rejected with a proper response.
1.If-modified
2.If-NotModified
3.If-Match
4.If-Nomatch

EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:
{real_name,[Path,AfterPath]} as defined in mod_alias(3) [page 66].

Exports the following EWSAPI interaction data, if possible:
{if_range,send_file} The conditions for the range request was not fullfilled. The
response must not be treated as a range request, instead it must be treated as a
ordinary get request.

Uses the following EWSAPI functions:
- mod_alias:parse/3 [page 67]

SEE ALSO

httpd(3) [page 40], mod_get(3) [page 65]
This module serves as a filter for authenticated requests handled in mod_auth. It provides possibility to restrict users from access for a specified amount of time if they fail to authenticate several times. It logs failed authentication as well as blocking of users, and it also calls a configurable call-back module when the events occur.

There is also an API to manually block, unblock and list blocked users or users, who have been authenticated within a configurable amount of time.

This module understands the following configuration directives:

- `<Directory>` [page 68]
- `SecurityDataFile` [page 109]
- `SecurityMaxRetries` [page 110]
- `SecurityBlockTime` [page 110]
- `SecurityFailExpireTime` [page 110]
- `SecurityAuthTimeout` [page 111]
- `SecurityCallbackModule` [page 111]

**DIRECTIVE: ”SecurityDataFile”**

**Syntax:** `SecurityDataFile` filename  
**Default:** None  
**Module:** `mod_security(3)` [page 109]  
**Context:** `<Directory>` [page 68]  
**Related:** `SecurityMaxRetries` [page 110], `SecurityBlockTime` [page 110], `SecurityFailExpireTime` [page 110], `SecurityAuthTimeout` [page 111], `SecurityCallbackModule` [page 111]

`SecurityDataFile` sets the name of the security modules for a directory. The filename can be either absolute or relative to the `ServerRoot`. This file is used to store persistent data for the `mod_security` module.

**Note:**  
Several directories can have the same `SecurityDataFile`. 
**DIRECTIVE: “SecurityMaxRetries”**

Syntax: `SecurityMaxRetries integer() | infinity`  
Default: 3  
Module: `mod_security(3)` [page 109]  
Context: `<Directory>` [page 68]  

`SecurityMaxRetries` specifies the maximum number of tries to authenticate a user has before he is blocked out. If a user successfully authenticates when he is blocked, he will receive a 403 (Forbidden) response from the server.

**Note:**  
For security reasons, failed authentications made by this user will return a message 401 (Unauthorized), even if the user is blocked.

**DIRECTIVE: “SecurityBlockTime”**

Syntax: `SecurityBlockTime integer() | infinity`  
Default: 60  
Module: `mod_security(3)` [page 109]  
Context: `<Directory>` [page 68]  

`SecurityBlockTime` specifies the number of minutes a user is blocked. After this amount of time, he automatically regains access.

**DIRECTIVE: “SecurityFailExpireTime”**

Syntax: `SecurityFailExpireTime integer() | infinity`  
Default: 30  
Module: `mod_security(3)` [page 109]  
Context: `<Directory>` [page 68]  

`SecurityFailExpireTime` specifies the number of minutes a failed user authentication is remembered. If a user authenticates after this amount of time, his previous failed authentications are forgotten.
DIRECTIVE: "SecurityAuthTimeout"

Syntax: SecurityAuthTimeout integer() | infinity
Default: 30
Module: mod_security(3) [page 109]
Context: <Directory> [page 68]
Related: SecurityDataFile [page 109], SecurityMaxRetries [page 110],
SecurityFailExpireTime [page 110], SecurityFailExpireTime [page 110],
SecurityCallbackModule [page 111]

SecurityAuthTimeout specifies the number of seconds a successful user authentication
is remembered. After this time has passed, the authentication will no longer be
reported by the list_auth_users [page 111] function.

DIRECTIVE: "SecurityCallbackModule"

Syntax: SecurityCallbackModule atom()
Default: -None-
Module: mod_security(3) [page 109]
Context: <Directory> [page 68]
Related: SecurityDataFile [page 109], SecurityMaxRetries [page 110],
SecurityFailExpireTime [page 110], SecurityFailExpireTime [page 110],
SecurityAuthTimeout [page 111]

SecurityCallbackModule specifies the name of a callback module. This module only
has one export, event/4 [page 113], which is called whenever a security event occurs.
Read the callback module [page 112] documentation to find out more.

Exports

list_auth_users(Port) -> Users | []
list_auth_users(Address, Port) -> Users | []
list_auth_users(Port, Dir) -> Users | []
list_auth_users(Address, Port, Dir) -> Users | []

Types:
- Port = integer()
- Address = (A,B,C,D) | string() | undefined
- Dir = string()
- Users = list() = [string()]

list_auth_users/1, list_auth_users/2 and list_auth_users/3 returns a list of users
that are currently authenticated. Authentications are stored for SecurityAuthTimeout
seconds, and are then discarded.

list_blocked_users(Port) -> Users | []
list_blocked_users(Address, Port) -> Users | []
list_blocked_users(Port, Dir) -> Users | []
list_blocked_users(Address, Port, Dir) -> Users | []
Types:

- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- Dir = string()
- Users = list() = [string()]

list_blocked_users/1, list_blocked_users/2 and list_blocked_users/3 returns a list of users that are currently blocked from access.

block_user(User, Port, Dir, Seconds) -> true | {error, Reason}
block_user(User, Address, Port, Dir, Seconds) -> true | {error, Reason}

Types:

- User = string()
- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- Dir = string()
- Seconds = integer() | infinity
- Reason = no_such_directory

block_user/4 and block_user/5 blocks the user User from the directory Dir for a specified amount of time.

unblock_user(User, Port) -> true | {error, Reason}
unblock_user(User, Address, Port) -> true | {error, Reason}
unblock_user(User, Port, Dir) -> true | {error, Reason}
unblock_user(User, Address, Port, Dir) -> true | {error, Reason}

Types:

- User = string()
- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- Dir = string()
- Reason = term()

unblock_user/2, unblock_user/3 and unblock_user/4 removes the user User from the list of blocked users for the Port (and Dir) specified.

The SecurityCallbackModule

The SecurityCallbackModule is a user written module that can receive events from the mod_security EWSAPI module. This module only exports one function, event/4 [page 113], which is described below.
Exports

\[
event(\text{What}, \text{Port}, \text{Dir}, \text{Data}) \rightarrow \text{ignored}
\]
\[
event(\text{What}, \text{Address}, \text{Port}, \text{Dir}, \text{Data}) \rightarrow \text{ignored}
\]

Types:
- \( \text{What} = \text{atom()} \)
- \( \text{Port} = \text{integer()} \)
- \( \text{Address} = \{A,B,C,D\} \) | \( \text{string()} \) \(<\text{v}>\text{Dir} = \text{string()} \)
- \( \text{What} = [\text{Info}] \)
- \( \text{Info} = \{\text{Name, Value}\} \)

\( \text{event/4 or event/5} \) is called whenever an event occurs in the \text{mod\_security EWSAPI} module (\( \text{event/4} \) is called if \( \text{Address} \) is undefined and \( \text{event/5} \) otherwise). The \text{What} argument specifies the type of event that has occurred, and should be one of the following reasons: \text{auth\_fail} (a failed user authentication), \text{user\_block} (a user is being blocked from access) or \text{user\_unblock} (a user is being removed from the block list).

\textbf{Note:}

Note that the \text{user\_unblock} event is not triggered when a user is removed from the block list explicitly using the \text{unblock\_user} function.

mod_trace

Erlang Module

This module is responsible for handling of TRACE requests. Trace is a new request method in HTTP/1.1. The intended use of trace requests is for testing. The body of the trace response is the request message that the responding Web server or proxy received.

EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:

SEE ALSO

httpd(3) [page 40],
Glossary

Gateway
A server which acts as an intermediary for some other server. Unlike a proxy, a gateway receives requests as if it were the origin server for the requested resource; the requesting client may not be aware that it is communicating with a gateway.

HTTP
Hypertext Transfer Protocol.

MIME
Multi-purpose Internet Mail Extensions.

Proxy
An intermediary program which acts as both a server and a client for the purpose of making requests on behalf of other clients.

RFC
A "Request for Comments" used as a proposed standard by IETF.
Index of Modules and Functions

Modules are typed in this way.
Functions are typed in this way.

<table>
<thead>
<tr>
<th>Module/Function</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>account/2</td>
<td>ftp</td>
<td>26</td>
</tr>
<tr>
<td>add_group_member/3</td>
<td>mod_auth</td>
<td>75</td>
</tr>
<tr>
<td>add_group_member/4</td>
<td>mod_auth</td>
<td>75</td>
</tr>
<tr>
<td>add_group_member/5</td>
<td>mod_auth</td>
<td>75</td>
</tr>
<tr>
<td>add_user/2</td>
<td>mod_auth</td>
<td>74</td>
</tr>
<tr>
<td>add_user/5</td>
<td>mod_auth</td>
<td>74</td>
</tr>
<tr>
<td>add_user/6</td>
<td>mod_auth</td>
<td>74</td>
</tr>
<tr>
<td>append/3</td>
<td>ftp</td>
<td>26</td>
</tr>
<tr>
<td>append_bin/3</td>
<td>ftp</td>
<td>26</td>
</tr>
<tr>
<td>append_chunk/2</td>
<td>ftp</td>
<td>26</td>
</tr>
<tr>
<td>append_chunk_end/1</td>
<td>ftp</td>
<td>27</td>
</tr>
<tr>
<td>append_chunk_start/2</td>
<td>ftp</td>
<td>26</td>
</tr>
<tr>
<td>block/0</td>
<td>httpd</td>
<td>41</td>
</tr>
<tr>
<td>block/1</td>
<td>httpd</td>
<td>41</td>
</tr>
<tr>
<td>block/2</td>
<td>httpd</td>
<td>41</td>
</tr>
<tr>
<td>block/3</td>
<td>httpd</td>
<td>41</td>
</tr>
<tr>
<td>block/4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>block_user/4</td>
<td>mod_security</td>
<td>112</td>
</tr>
<tr>
<td>block_user/5</td>
<td>mod_security</td>
<td>112</td>
</tr>
<tr>
<td>cancel_request/1</td>
<td>http</td>
<td>36</td>
</tr>
<tr>
<td>cd/2</td>
<td>ftp</td>
<td>27</td>
</tr>
<tr>
<td>check_enum/2</td>
<td>httpd_conf</td>
<td>45</td>
</tr>
<tr>
<td>clean/1</td>
<td>httpd_conf</td>
<td>45</td>
</tr>
<tr>
<td>close/1</td>
<td>ftp</td>
<td>27</td>
</tr>
<tr>
<td>convert_request_date/1</td>
<td>httpd_util</td>
<td>57</td>
</tr>
<tr>
<td>cookie_header/1</td>
<td>http</td>
<td>38</td>
</tr>
<tr>
<td>create_etag/1</td>
<td>httpd_util</td>
<td>57</td>
</tr>
<tr>
<td>custom_clean/3</td>
<td>httpd_conf</td>
<td>45</td>
</tr>
<tr>
<td>day/1</td>
<td>httpd_util</td>
<td>57</td>
</tr>
<tr>
<td>decode/1</td>
<td>http_base64</td>
<td>39</td>
</tr>
<tr>
<td>decode_base64/1</td>
<td>httpd_util</td>
<td>57</td>
</tr>
<tr>
<td>decode_hex/1</td>
<td>httpd_util</td>
<td>57</td>
</tr>
<tr>
<td>default_index/2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Inets 117
Index of Modules and Functions

mod_alias, 66
delete/2
ftp, 27
delete_group/2
mod_auth, 77
delete_group/4
mod_auth, 77
delete_group_member/3
mod_auth, 76
delete_group_member/4
mod_auth, 76
delete_group_member/5
mod_auth, 76
delete_user/2
mod_auth, 74
delete_user/3
mod_auth, 74
delete_user/4
mod_auth, 74
deliver/2
mod_esi, 92
deliver/3
httpd_socket, 56
encode/1
http_base64, 39
encode_base64/1
httpd_util, 58
env/3
mod_cgi, 81
error_log/5
mod_disklog, 86
mod_log, 106
event/4
mod_security, 113
event/5
mod_security, 113
flatlength/1
httpd_util, 58
formaterror/1
ftp, 27
ftp
account/2, 26
append/3, 26
append_bin/3, 26
append_chunk/2, 26
append_chunk_end/1, 27
append_chunk_start/2, 26
cd/2, 27
close/1, 27
delete/2, 27
formaterror/1, 27
cd/2, 27
lpwd/1, 27
ls/2, 28
mkdir/2, 28
nlist/2, 28
open/2, 28
open/3, 28
pwd/1, 30
quote/2, 33
recv/3, 30
recv_bin/2, 30
recv_chunk/1, 31
recv_chunk_start/2, 31
rename/3, 31
rmdir/2, 31
send/3, 31
send_bin/3, 32
send_chunk/2, 32
send_chunk_end/1, 32
send_chunk_start/2, 32
type/2, 32
user/3, 32
user/4, 33
get_user/2
mod_auth, 74
get_user/3
mod_auth, 74
get_user/4
mod_auth, 74
getBrowser/1
mod_browser, 79
header/2
httpd_util, 58
header/3
httpd_util, 58
header/4
httpd_util, 58
hexlist_to_integer/1
httpd_util, 58
## Index of Modules and Functions

### http
- cancel_request/1, 36
- cookie_header/1, 38
- request/1, 36
- request/4, 36
- set_options/1, 37
- verify_cookie/2, 38

### http_base64
- decode/1, 39
- encode/1, 39

### httpd
- block/0, 41
- block/1, 41
- block/2, 41
- block/3, 41
- block/4, 41
- Module:do/1, 43
- Module:load/2, 43
- Module:remove/1, 44
- Module:store/3, 44
- parse_query/1, 42
- restart/0, 40
- restart/1, 40
- restart/2, 40
- start/0, 40
- start/1, 40
- start_link/0, 40
- start_link/1, 40
- stop/0, 41
- stop/1, 41
- stop/2, 41
- unblock/0, 42
- unblock/1, 42
- unblock/2, 42

### httpd_conf
- check_enum/2, 45
- clean/1, 45
- custom_clean/3, 45
- is_directory/1, 45
- is_file/1, 46
- make_integer/1, 46

### httpd_socket
- deliver/3, 56
- peername/2, 56
- resolve/0, 56

### httpd_util
- convert_request_date/1, 57
- create_etag/1, 57
- day/1, 57
- decode_base64/1, 57
- decode_hex/1, 57
- encode_base64/1, 58
- flatlength/1, 58
- header/2, 58
- header/3, 58
- header/4, 58
- hexlist_to_integer/1, 58
- integer_tohexlist/1, 58
- key1search/2, 59
- key1search/3, 59
- lookup/2, 59
- lookup/3, 59
- lookup_mime/2, 59
- lookup_mime/3, 59
- lookup_mime_default/2, 59
- lookup_mime_default/3, 59
- message/3, 60
- month/1, 60
- multi_lookup/2, 60
- reason_phrase/1, 60
- rfc1123_date/0, 61
- rfc1123_date/6, 61
- split/3, 61
- split_path/1, 61
- split_script_path/1, 61
- strip/1, 61
- suffix/1, 62
- to_lower/1, 62
- to_upper/1, 62

- integer_tohexlist/1
  - httpd_util, 58

- is_directory/1
  - httpd_conf, 45

- is_file/1
  - httpd_conf, 46

- key1search/2
  - httpd_util, 59

- key1search/3
  - httpd_util, 59

- lcd/2
  - ftp, 27

- list_auth_users/1
  - mod_security, 111

- list_auth_users/2
  - mod_security, 111

- list_auth_users/3
<table>
<thead>
<tr>
<th>Module</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod_security</td>
<td>list_blocked_users/1</td>
</tr>
<tr>
<td></td>
<td>list_blocked_users/2</td>
</tr>
<tr>
<td></td>
<td>list_blocked_users/3</td>
</tr>
<tr>
<td></td>
<td>list_group_members/2</td>
</tr>
<tr>
<td></td>
<td>list_group_members/3</td>
</tr>
<tr>
<td></td>
<td>list_group_members/4</td>
</tr>
<tr>
<td></td>
<td>list_groups/1</td>
</tr>
<tr>
<td></td>
<td>list_groups/2</td>
</tr>
<tr>
<td></td>
<td>list_groups/3</td>
</tr>
<tr>
<td></td>
<td>list_users/1</td>
</tr>
<tr>
<td></td>
<td>list_users/3</td>
</tr>
<tr>
<td></td>
<td>lookup/2</td>
</tr>
<tr>
<td></td>
<td>lookup/3</td>
</tr>
<tr>
<td></td>
<td>lookup_mime/2</td>
</tr>
<tr>
<td></td>
<td>lookup_mime/3</td>
</tr>
<tr>
<td></td>
<td>lookup_mime_default/2</td>
</tr>
<tr>
<td></td>
<td>lookup_mime_default/3</td>
</tr>
<tr>
<td></td>
<td>lpwd/1</td>
</tr>
<tr>
<td></td>
<td>ls/2</td>
</tr>
<tr>
<td></td>
<td>make_integer/1</td>
</tr>
<tr>
<td></td>
<td>message/3</td>
</tr>
<tr>
<td>httpd util</td>
<td>mkdir/2</td>
</tr>
<tr>
<td></td>
<td>ftp</td>
</tr>
<tr>
<td>mod_alias</td>
<td>default_index/2</td>
</tr>
<tr>
<td></td>
<td>path/3</td>
</tr>
<tr>
<td></td>
<td>real_name/3</td>
</tr>
<tr>
<td></td>
<td>real_script_name/3</td>
</tr>
<tr>
<td>mod_auth</td>
<td>add_group_member/3</td>
</tr>
<tr>
<td></td>
<td>add_group_member/4</td>
</tr>
<tr>
<td></td>
<td>add_group_member/5</td>
</tr>
<tr>
<td></td>
<td>add_user/2</td>
</tr>
<tr>
<td></td>
<td>add_user/5</td>
</tr>
<tr>
<td></td>
<td>add_user/6</td>
</tr>
<tr>
<td></td>
<td>delete_group/2</td>
</tr>
<tr>
<td></td>
<td>delete_group/4</td>
</tr>
<tr>
<td></td>
<td>delete_group_member/3</td>
</tr>
<tr>
<td></td>
<td>delete_group_member/4</td>
</tr>
<tr>
<td></td>
<td>delete_group_member/5</td>
</tr>
<tr>
<td></td>
<td>delete_user/2</td>
</tr>
<tr>
<td></td>
<td>delete_user/3</td>
</tr>
<tr>
<td></td>
<td>delete_user/4</td>
</tr>
<tr>
<td></td>
<td>get_user/2</td>
</tr>
<tr>
<td></td>
<td>get_user/3</td>
</tr>
<tr>
<td></td>
<td>get_user/4</td>
</tr>
<tr>
<td></td>
<td>list_group_members/2</td>
</tr>
<tr>
<td></td>
<td>list_group_members/3</td>
</tr>
<tr>
<td></td>
<td>list_group_members/4</td>
</tr>
<tr>
<td></td>
<td>list_groups/1</td>
</tr>
<tr>
<td></td>
<td>list_groups/2</td>
</tr>
<tr>
<td></td>
<td>list_groups/3</td>
</tr>
<tr>
<td></td>
<td>list_users/1</td>
</tr>
<tr>
<td></td>
<td>list_users/3</td>
</tr>
<tr>
<td></td>
<td>update_password/5</td>
</tr>
<tr>
<td></td>
<td>update_password/6</td>
</tr>
<tr>
<td>mod_browser</td>
<td>getBrowser/1</td>
</tr>
<tr>
<td>mod_cgi</td>
<td>env/3</td>
</tr>
<tr>
<td></td>
<td>status_code/1</td>
</tr>
<tr>
<td>mod_disk_log</td>
<td>error_log/5</td>
</tr>
<tr>
<td></td>
<td>security_log/2</td>
</tr>
<tr>
<td>mod_esi</td>
<td>deliver/2</td>
</tr>
<tr>
<td></td>
<td>Module:Function/2</td>
</tr>
<tr>
<td></td>
<td>Module:Function/3</td>
</tr>
<tr>
<td>mod_log</td>
<td></td>
</tr>
</tbody>
</table>
error_log/5, 106

mod_security
  block_user/4, 112
  block_user/5, 112
  event/4, 113
  event/5, 113
  list_auth_users/1, 111
  list_auth_users/2, 111
  list_auth_users/3, 111
  list_blocked_users/1, 111
  list_blocked_users/2, 111
  list_blocked_users/3, 111
  unblock_user/2, 112
  unblock_user/3, 112
  unblock_user/4, 112

Module:do/1
  httpd, 43

Module:Function/2
  mod_esi, 93

Module:Function/3
  mod_esi, 93

Module:load/2
  httpd, 43

Module:remove/1
  httpd, 44

Module:store/3
  httpd, 44

month/1
  httpd_util, 60

multi_lookup/2
  httpd_util, 60

nlist/2
  ftp, 28

open/2
  ftp, 28

open/3
  ftp, 28

parse_query/1
  httpd, 42

path/3
  mod_alias, 66

peername/2
  httpd_socket, 56

pwd/1
  ftp, 30
  http, 30

quote/2
  ftp, 33

real_name/3
  mod_alias, 67

real_script_name/3
  mod_alias, 67

reason_phrase/1
  httpd_util, 60

recv/3
  ftp, 30

recv_bin/2
  ftp, 30

recv_chunk/1
  ftp, 31

recv_chunk_start/2
  ftp, 31

rename/3
  ftp, 31

request/1
  http, 36

request/4
  http, 36

resolve/0
  httpd_socket, 56

restart/0
  httpd, 40

restart/1
  httpd, 40

restart/2
  httpd, 40

rfc1123_date/0
  httpd_util, 61

rfc1123_date/6
  httpd_util, 61

rmdir/2
  ftp, 31

security_log/2
  mod_disk_log, 87

send/3
Index of Modules and Functions

ftp, 31
send_bin/3
ftp, 32
send_chunk/2
ftp, 32
send_chunk_end/1
ftp, 32
send_chunk_start/2
ftp, 32
set_options/1
http, 37
split/3
httpd_util, 61
split_path/1
httpd_util, 61
split_script_path/1
httpd_util, 61
start/0
httpd, 40
start/1
httpd, 40
start_link/0
httpd, 40
start_link/1
httpd, 40
status_code/1
mod_cgi, 81
stop/0
httpd, 41
stop/1
httpd, 41
stop/2
httpd, 41
strip/1
httpd_util, 61
suffix/1
httpd_util, 62
to_lower/1
httpd_util, 62
to_upper/1
httpd_util, 62
type/2
ftp, 32
unblock/0
httpd, 42
unblock/1
httpd, 42
unblock/2
httpd, 42
unblock_user/2
mod_security, 112
unblock_user/3
mod_security, 112
unblock_user/4
mod_security, 112
update_password/5
mod_auth, 77
update_password/6
mod_auth, 77
user/3
ftp, 32
user/4
ftp, 33
verify_cookie/2
http, 38