

Internet Services Application (INETS)

version 2.5

Mattias Nilsson

1997-07-16

Typeset in \LaTeX from SGML source using the DOCBUILDER 3.0 Document System.

Contents

1	INETS Reference Manual	1
1.1	inets (Application)	9
1.2	ftp (Module)	10
1.3	httpd (Module)	16
1.4	httpd_conf (Module)	25
1.5	httpd_core (Module)	27
1.6	httpd_socket (Module)	33
1.7	httpd_util (Module)	35
1.8	mod_actions (Module)	40
1.9	mod_alias (Module)	42
1.10	mod_auth (Module)	45
1.11	mod_cgi (Module)	53
1.12	mod_dir (Module)	55
1.13	mod_disk_log (Module)	56
1.14	mod_esi (Module)	59
1.15	mod_include (Module)	63
1.16	mod_log (Module)	66
1.17	mod_security (Module)	68
	List of Terms	73

INETs Reference Manual

Short Summaries

- Application **inets** [page 9] – The Internet Services Application
- Erlang Module **ftp** [page 10] – A File Transfer Protocol client
- Erlang Module **httpd** [page 16] – An implementation of an HTTP 1.0 compliant Web server, as defined in RFC 1945.
- Erlang Module **httpd_conf** [page 25] – Configuration utility functions to be used by the EWSAPI programmer.
- Erlang Module **httpd_core** [page 27] – The core functionality of the Web server.
- Erlang Module **httpd_socket** [page 33] – Communication utility functions to be used by the EWSAPI programmer.
- Erlang Module **httpd_util** [page 35] – Miscellaneous utility functions to be used when implementing EWSAPI modules.
- Erlang Module **mod_actions** [page 40] – Filetype/method-based script execution.
- Erlang Module **mod_alias** [page 42] – This module creates aliases and redirections.
- Erlang Module **mod_auth** [page 45] – User authentication using text files, dets or mnesia database.
- Erlang Module **mod_cgi** [page 53] – Invoking of CGI scripts.
- Erlang Module **mod_dir** [page 55] – Basic directory handling.
- Erlang Module **mod_disk_log** [page 56] – Standard logging using the "Common Logfile Format" and `disk_log(3)`.
- Erlang Module **mod_esi** [page 59] – Efficient Erlang Scripting.
- Erlang Module **mod_include** [page 63] – Server-parsed documents.
- Erlang Module **mod_log** [page 66] – Standard logging using the "Common Logfile Format" and text files.
- Erlang Module **mod_security** [page 68] – Security Audit and Trailing Functionality

inets

No functions are exported

ftp

The following functions are exported:

- `cd(Pid, Dir) -> ok | {error, Reason}`
[page 11] Change remote working directory.
- `close(Pid) -> ok`
[page 11] End ftp session.
- `delete(Pid, File) -> ok | {error, Reason}`
[page 11] Delete a file at the remote server..
- `formaterror(Tag) -> string()`
[page 11] Return error diagnostics.
- `lcd(Pid, Dir) -> ok | {error, Reason}`
[page 11] Change local working directory.
- `lpwd(Pid) -> {ok, Dir}`
[page 11] Get local current working directory.
- `ls(Pid [, Dir]) -> {ok, Listing} | {error, Reason}`
[page 11] List contents of remote directory.
- `mkdir(Pid, Dir) -> ok | {error, Reason}`
[page 12] Create remote directory.
- `nlist(Pid [, Dir]) -> {ok, Listing} | {error, Reason}`
[page 12] List contents of remote directory.
- `open(Host [, Flags]) -> {ok, Pid} | {error, Reason}`
[page 12] Start an ftp client.
- `pwd(Pid) -> {ok, Dir} | {error, Reason}`
[page 13] Get remote current working directory.
- `recv(Pid, RemoteFile [, LocalFile]) -> ok | {error, Reason}`
[page 13] Transfer file from remote server.
- `recv_bin(Pid, RemoteFile) -> {ok, Bin} | {error, Reason}`
[page 13] Transfer file from remote server as a binary.
- `rename(Pid, Old, New) -> ok | {error, Reason}`
[page 13] Rename a file at the remote server..
- `rmdir(Pid, Dir) -> ok | {error, Reason}`
[page 13] Remove a remote directory.
- `send(Pid, LocalFile [, RemoteFile]) -> ok | {error, Reason}`
[page 13] Transfer file to remote server.
- `send_bin(Pid, Bin, RemoteFile) -> ok | {error, Reason}`
[page 14] Transfer a binary into a remote file.
- `send_chunk(Pid, Bin) -> ok | {error, Reason}`
[page 14] Write a chunk to the remote file.
- `send_chunk_end(Pid) -> ok | {error, Reason}`
[page 14] Stop transfer of chunks.
- `send_chunk_start(Pid, File) -> ok | {error, Reason}`
[page 14] Start transfer of file chunks.
- `type(Pid, Type) -> ok | {error, Reason}`
[page 14] Set transfer type to ascii or binary.
- `user(Pid, User, Password) -> ok | {error, Reason}`
[page 14] User login.

httpd

The following functions are exported:

- `start()`
[page 18] Starts a server as specified in the given config file.
- `start(ConfigFile) -> ServerRet`
[page 18] Starts a server as specified in the given config file.
- `restart()`
[page 18] Restarts a server which listens to a specific port.
- `restart(Port) -> ServerRet`
[page 18] Restarts a server which listens to a specific port.
- `stop()`
[page 18] Stops a server which listens to a specific port.
- `stop(Port) -> ServerRet`
[page 19] Stops a server which listens to a specific port.
- `parse_query(QueryString) -> ServerRet`
[page 19] Parses incoming data to `erl` and `eval` scripts.

httpd_conf

The following functions are exported:

- `check_enum(EnumString,ValidEnumStrings) -> Result`
[page 25] Checks if string is a valid enumeration.
- `clean(String) -> Stripped`
[page 25] Removes leading and/or trailing white spaces.
- `custom_clean(String,Before,After) -> Stripped`
[page 25] Removes leading and/or trailing white spaces and custom characters.
- `is_directory(FilePath) -> Result`
[page 25] Checks if a file path is a directory.
- `is_file(FilePath) -> Result`
[page 26] Checks if a file path is a regular file.
- `make_integer(String) -> Result`
[page 26] Returns 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 33] Sends binary data over a socket in 2kB packets.
- `peername(SocketType,Socket) -> {Port,IPaddress}`
[page 33] Returns the port and IP-address of the remote socket.
- `resolve() -> HostName`
[page 33] Returns the official name of the current host.

httpd_util

The following functions are exported:

- `decode_base64(Base64String) -> ASCIIString`
[page 35] Converts a base64 encoded string to a plain ascii string.
- `decode_hex(HexValue) -> DecValue`
[page 35] Converts a hex value into its decimal equivalent.
- `day(NthDayOfWeek) -> DayOfWeek`
[page 35] Converts the day of the week (integer [1-7]) to an abbreviated string.
- `encode_base64(ASCIIString) -> Base64String`
[page 35] Converts an ASCII string to a Base64 encoded string.
- `header(StatusCode,Date)`
[page 36] Generates a HTTP 1.0 header.
- `header(StatusCode,MimeType,Date) -> HTTPHeader`
[page 36] Generates a HTTP 1.0 header.
- `flatlength(NestedList) -> Size`
[page 36] Computes the size of a possibly nested list.
- `key1search(TupleList,Key)`
[page 36] Searches a list of key-value tuples for a tuple whose first element is a key.
- `key1search(TupleList,Key,Undefined) -> Result`
[page 36] Searches a list of key-value tuples for a tuple whose first element is a key.
- `lookup(ETSTable,Key) -> Result`
[page 36] Used to extract the first value associated with a key in an ETS table.
- `lookup(ETSTable,Key,Undefined) -> Result`
[page 36] Used to extract the first value associated with a key in an ETS table.
- `lookup_mime(ConfigDB,Suffix)`
[page 36] Returns the mime type associated with a specific file suffix.
- `lookup_mime(ConfigDB,Suffix,Undefined) -> MimeType`
[page 36] Returns the mime type associated with a specific file suffix.
- `lookup_mime_default(ConfigDB,Suffix)`
[page 37] Returns the mime type associated with a specific file suffix or the value of the DefaultType.
- `lookup_mime_default(ConfigDB,Suffix,Undefined) -> MimeType`
[page 37] Returns the mime type associated with a specific file suffix or the value of the DefaultType.

- `message(StatusCode,PhraseArgs,ConfigDB) -> Message`
[page 37] Returns an informative HTTP 1.0 status string in HTML.
- `month(NthMonth) -> Month`
[page 37] Converts the month as an integer (1-12) to an abbreviated string.
- `multi_lookup(ETSTable,Key) -> Result`
[page 38] Used to extract the values associated with a key in a ETS table.
- `reason_phrase(StatusCode) -> Description`
[page 38] Returns the description of an HTTP 1.0 status code.
- `rfc1123_date() -> RFC1123Date`
[page 38] Returns the current date in RFC 1123 format.
- `split(String,RegExp,N) -> SplitRes`
[page 38] Splits a string in N chunks using a regular expression.
- `split_script_path(RequestLine) -> Splitted`
[page 38] Splits a RequestLine in a file reference to an executable and a QueryString or a PathInfo string.
- `split_path(RequestLine) -> {Path,QueryStringOrPathInfo}`
[page 38] Splits a RequestLine in a file reference and a QueryString or a PathInfo string.
- `suffix(FileName) -> Suffix`
[page 39] Extracts the file suffix from a given filename.
- `to_lower(String) -> ConvertedString`
[page 39] Converts upper-case letters to lower-case.
- `to_upper(String) -> ConvertedString`
[page 39] Converts lower-case letters to upper-case.

mod_actions

No functions are exported

mod_alias

The following functions are exported:

- `default_index(ConfigDB,Path) -> NewPath`
[page 43] Returns a new path with the default resource or file appended.
- `path(Data,ConfigDB,RequestURI) -> Path`
[page 44] Returns the actual file path to a URL.
- `real_name(ConfigDB,RequestURI,Aliases) -> Ret`
[page 44] Expands a request uri using Alias config directives.
- `real_script_name(ConfigDB,RequestURI,ScriptAliases) -> Ret`
[page 44] Expands a request uri using ScriptAlias config directives.

mod_auth

The following functions are exported:

- `add_user(Username, Password, UserData, Port, Dir) -> true | {error, Reason}`
[page 50] Add a user to the user database.
- `delete_user(Username, Port, Dir) -> true | {error, Reason}`
[page 50] Delete a user from the user database.
- `get_user(Username, Port, Dir) -> {ok, #httpd_user} | {error, Reason}`
[page 50] Delete a user from the user database.
- `list_users(Port, Dir) -> {ok, Users}`
[page 51] List users in the user database.
- `add_group_member(GroupName, Username, Port, Dir) -> true | {error, Reason}`
[page 51] Add a user to a group.
- `delete_group_member(GroupName, Username, Port, Dir) -> true | {error, Reason}`
[page 51] Remove a user from a group.
- `list_group_members(GroupName, Port, Dir) -> {ok, Users} | {error, Reason}`
[page 51] List the members of a group.
- `list_groups(Port, Dir) -> {ok, Groups} | {error, Reason}`
[page 51] List all the groups.
- `delete_group(GroupName, Port, Dir) -> true | {error, Reason}`
[page 52] List all the groups.

mod_cgi

The following functions are exported:

- `env(Info,Script,AfterScript) -> EnvString`
[page 54] Returns a CGI-1.1 environment variable string to be used by `open_port/2`.
- `status_code(CGIOutput) -> {ok,StatusCode} | {error,Reason}`
[page 54] Parses 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 58] Logs an error in the error log file.

mod_esi

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 67] Logs an error in the a log file.

mod_security

The following functions are exported:

- `list_auth_users(Port) -> Users | []`
[page 70] Lists users that have authenticated within the SecurityAuthTimeout time for a given port number (and directory, if specified).
- `list_auth_users(Port, Dir) -> Users | []`
[page 70] Lists users that have authenticated within the SecurityAuthTimeout time for a given port number (and directory, if specified).
- `list_blocked_users(Port) -> Users | []`
[page 70] Lists users that are currently blocked from access to a specified port number.
- `list_blocked_users(Port, Dir) -> Users | []`
[page 70] Lists users that are currently blocked from access to a specified port number.
- `block_user(User, Port, Dir, Seconds) -> true | {error, no_such_directory}`
[page 71] Blocked user from access to a directory for a certain amount of time.

- `unblock_user(User, Port) -> true | {error, Reason}`
[page 71] Remove a blocked user from the block list
- `unblock_user(User, Port, Dir) -> true | {error, Reason}`
[page 71] Remove a blocked user from the block list
- `event(What, Port, Dir, Data) -> ignored`
[page 71] This function is called whenever an event occurs in `mod_security`

inets (Application)

The Internet Services Application (INETS) is a container for Internet clients and servers. Currently, an HTTP server and an FTP client has been incorporated in INETS. The HTTP server is an efficient implementation of *HTTP* 1.0 as defined in *RFC* 1945, namely a Web server.

Configuration

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.conf`) starting two HTTP servers typically looks as follows:

```
[{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 `httpd(3)` [page 16].

`inets.conf` can be tested by copying the example server root to a specific installation directory, as described in `httpd(3)` [page 18]. The example below shows a manual start of an Erlang node, using `inets.conf`, and the start of two HTTP servers listening listen 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, <0.24.0>}
```

SEE ALSO

`httpd(3)` [page 16]

ftp (Module)

The `ftp` module implements a client for file transfer according to a subset of the File Transfer Protocol (see *RFC 959*).

The following is a simple example of an ftp session, where the user `guest` with password `password` logs on to the remote host `agner.krarup.erlang.com`, 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> {ok, Pid} = ftp:open("agner.krarup.erlang.com").
{ok, <0.22.0>}
2> ftp:user(Pid, "guest", "password").
ok
3> ftp:pwd(Pid).
{ok, "/home/guest"}
4> ftp:cd(Pid, "appl/examples").
ok
5> ftp:lpwd(Pid).
{ok, "/home/fred"}.
6> ftp:lcd(Pid, "/home/eproj/examples").
ok
7> ftp:recv(Pid, "appl.erl").
ok
8> ftp:close(Pid).
ok
```

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 contiguous parts of a file to be stored in a remote file (see `send_chunk_start/2`, `send_chunk/2` and `send_chunk_end/1`).

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.

Exports

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

Types:

- Pid = pid()
- Dir = string()
- Reason = epath | elogin | econn

Changes the working directory at the remote server to Dir.

`close(Pid) -> ok`

Types:

- Pid = pid()

Ends the ftp session.

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

Types:

- Pid = pid()
- File = string()
- Reason = epath | elogin | econn

Deletes the file File at the remote server.

`formaterror(Tag) -> string()`

Types:

- Tag = {error, atom()} | atom()

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

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

Types:

- Pid = pid()
- Dir = string()
- Reason = epath

Changes the working directory to Dir for the local client.

`lpwd(Pid) -> {ok, Dir}`

Types:

- Pid = pid()

Returns the current working directory at the local client.

`ls(Pid [, Dir]) -> {ok, Listing} | {error, Reason}`

Types:

- Pid = pid()
- Dir = string()
- Listing = string()
- Reason = epath | elogin | econn

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

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

Types:

- Pid = pid()
- Dir = string()
- Reason = epath | elogin | econn

Creates the directory `Dir` at the remote server.

`nlist(Pid [, Dir]) -> {ok, Listing} | {error, Reason}`

Types:

- Pid = pid()
- Dir = string()
- Listing = string()
- Reason = epath | elogin | econn

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

`open(Host [, Flags]) -> {ok, Pid} | {error, Reason}`

Types:

- Host = string() | ip_address()
- ip_address() = {byte(), byte(), byte(), byte()}
- byte() = 0 | 1 | ... | 255
- Flags = [Flag]
- Flag = verbose
- Pid = pid()
- Reason = ehost

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 (e.g. {150, 236, 14, 136}).

If `Flags` is set, response messages from the remote server will be written to standard output.

The file transfer type is set to binary when the session is opened.

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

The return value `Pid` is used as a reference to the newly created ftp client in all other functions. The ftp client process is linked to the caller.

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

Types:

- `Pid = pid()`
- `Reason = elogin | econn`

Returns the current working directory at the remote server.

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

Types:

- `Pid = pid()`
- `RemoteFile = LocalFile = string()`
- `Reason = epath | elogin | econn`

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`.

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

Types:

- `Pid = pid()`
- `Bin = binary()`
- `RemoteFile = string()`
- `Reason = epath | elogin | econn`

Transfers the file `RemoteFile` from the remote server and receives it as a binary.

`rename(Pid, Old, New) -> ok | {error, Reason}`

Types:

- `Pid = pid()`
- `CurrFile = NewFile = string()`
- `Reason = epath | elogin | econn`

Renames `Old` to `New` at the remote server.

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

Types:

- `Pid = pid()`
- `Dir = string()`
- `Reason = epath | elogin | econn`

Removes directory `Dir` at the remote server.

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

Types:

- `Pid = pid()`

- LocalFile = RemoteFile = string()
- Reason = epath | elogin | econn

Transfers the file LocalFile to the remote server. If RemoteFile is specified, the name of the remote file is set to RemoteFile; otherwise the name is set to LocalFile.

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

Types:

- Pid = pid()
- Bin = binary()
- RemoteFile = string()
- Reason = epath | elogin | enotbinary | econn

Transfers the binary Bin into the file RemoteFile at the remote server.

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

Types:

- Pid = pid()
- Bin = binary()
- Reason = elogin | chunk | enotbinary | econn

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

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

Types:

- Pid = pid()
- Reason = elogin | chunk | econn

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.

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

Types:

- Pid = pid()
- File = string()
- Reason = epath | elogin | econn

Start transfer of chunks into the file File at the remote server.

```
type(Pid, Type) -> ok | {error, Reason}
```

Types:

- Pid = pid()
- Type = ascii | binary
- Reason = etype | elogin | econn

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

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

Types:

- Pid = pid()
- User = Password = string()
- Reason = euser | econn

Performs login of User with Password.

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.

`enotbinary` 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.

SEE ALSO

`file`, `filename`, J. Postel and J. Reynolds: File Transfer Protocol (RFC 959).

httpd (Module)

HTTP (Hypertext Transfer Protocol) is an application-level protocol with the lightness and speed necessary for distributed, collaborative and hyper-media information systems. The `httpd` module handles HTTP 1.0 as described in *RFC 1945* with a few exceptions such as *Gateway* and *Proxy* functionality. The same is true for servers written by NCSA and others.

The server implements numerous features such as SSL [page 27] (Secure Sockets Layer), ESI [page 59] (Erlang Scripting Interface), CGI [page 53] (Common Gateway Interface), User Authentication [page 45] (using Mnesia, dets or plain text database), Common Logfile Format (with [page 56] or without [page 66] `disk_log(3)` support), URL Aliasing [page 42], Action Mappings [page 40], Directory Listings [page 55] and SSI [page 63] (Server-Side Includes).

The configuration [page 16] of the server is done using Apache¹-style run-time configuration directives. The goal is to be plug-in compatible with Apache but with enhanced fault-tolerance, scalability and load-balancing characteristics.

All server functionality has been implemented using an especially crafted server API; EWSAPI [page 19] (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.

RUN-TIME CONFIGURATION

All functionality in the server can be configured using Apache-style run-time 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:

- Action [page 40]
- Alias [page 42]
- allow [page 48]
- deny [page 49]
- AuthName [page 48]
- AuthGroupFile [page 47]
- AuthUserFile [page 46]
- BindAddress [page 28]
- DefaultType [page 28]

¹URL: <http://www.apache.org>

²In Windows: `%INETs_ROOT%\examples\server_root\conf\`. In UNIX: `$INETs_ROOT/examples/server_root/conf/`.

- <Directory> [page 45]
- DirectoryIndex [page 42]
- DocumentRoot [page 29]
- ErlScriptAlias [page 61]
- ErrorLog [page 66]
- ErrorDiskLog [page 56]
- ErrorDiskLogSize [page 57]
- EvalScriptAlias [page 62]
- MaxClients [page 29]
- Modules [page 29]
- Port [page 30]
- require [page 49]
- ServerAdmin [page 30]
- ServerName [page 30]
- ServerRoot [page 31]
- Script [page 40]
- ScriptAlias [page 43]
- SocketType [page 31]
- SSLCertificateFile [page 31]
- SSLCertificateKeyFile [page 31]
- SSLVerifyClient [page 32]
- KeepAlive [page 32]
- KeepAliveTimeout [page 32]
- TransferLog [page 67]
- TransferDiskLog [page 57]
- TransferDiskLogSize [page 57]

EWSAPI MODULES

All server functionality below has been implemented using EWSAPI (Erlang Web Server API) modules. The following modules all have separate manual pages (`mod_cgi(3)`, `mod_auth(3)`, ...):

httpd_core [page 27] Core features.

mod_actions [page 40] Filetype/method-based script execution.

mod_alias [page 42] Aliases and redirects.

mod_auth [page 45] User authentication using text files, mnesia or dets

mod_cgi [page 53] Invoking of CGI scripts.

mod_dir [page 55] Basic directory handling.

mod_esi [page 59] Efficient Erlang Scripting.

mod_get HTTP GET Method

mod_head HTTP HEAD Method

mod_include [page 63] Server-parsed documents.

mod_log [page 66] Standard logging in the Common Logfile Format using text files.

mod_disk_log [page 56] Standard logging in the Common Logfile Format using `disk_log(3)`.

The Modules [page 29] config directive can be used to alter the server behavior, that is to 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 [page 23] section below.

Exports

`start()`

`start(ConfigFile) -> ServerRet`

Types:

- `ConfigFile = string()`
- `ServerRet = {ok,Pid} | ignore | {error,EReason} | {stop,SReason}`
- `Pid = pid()`
- `EReason = {already_started, Pid} | term()`
- `SReason = string()`

`start/1` starts a server as specified in the given `ConfigFile`. The `ConfigFile` supports a number of config directives specified below.

`start/0` starts a server as specified in a hard-wired config file, that is `start("/var/tmp/server_root/conf/8888.conf")`. Before utilizing `start/0`, copy the example server root³ to a specific installation directory⁴ and you have a server running in no time.

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")`.

`restart()`

`restart(Port) -> ServerRet`

Types:

- `Port = int()`
- `ServerRet = ok | not_started`

`restart/1` restarts a server which listens to a specific `Port` and reloads its config file.

`restart/0` restarts a server which listens to port 8888, that is `restart(8888)`.

`stop()`

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

⁴In Windows: X:\var\tmp\. In UNIX: /var/tmp/.

`stop(Port) -> ServerRet`

Types:

- `Port = int()`
- `ServerRet = ok | not_started`

`stop/1` stops a server which listens to a specific `Port`. `stop/0` stops a server which listens to port `8888`, that is `stop(8888)`.

`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_esi(3)` [page 59]) as defined in the standard URL format, that is '+' becomes 'space' and decoding of hexadecimal characters (`%xx`).

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 2109⁵ implementation.

Warning:

The current implementation of EWSAPI is under review and feedback is welcomed.

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 53] or ESI [page 59] 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 modules⁶ `mod_*.erl` and note how to they are configured in the example config directory⁷.

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 (See EWSAPI Module Configuration [page 21] below).

⁶In Windows: `%INETS_ROOT%\src\`. In UNIX: `$INETS_ROOT/src/`.

⁷In Windows: `%INETS_ROOT%\examples\server_root\conf\`. In UNIX: `$INETS_ROOT/examples/server_root/conf/`.

A module can choose to export functions to be used by other modules in the EWSAPI Module Sequence (See Modules [page 29] 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 (See EWSAPI Module Interaction [page 23] below).

Note:

The server executes `do/1` (using `apply/1`) for each module listed in the Modules [page 29] config directive. `do/1` takes the record `mod` as an argument, as described below. See `httpd.hrl`⁸:

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

The fields of the `mod` record has the following meaning:

`data` Type `[{InteractionKey, InteractionValue}]` is used to propagate data between modules (See EWSAPI Module Interaction [page 23] below). Depicted `interaction_data()` in function type declarations.

`socket_type` Type `ip_comm | ssl`, that is the socket type.

`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" | "DELETE" | "PUT"`, that is the HTTP method.

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

`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}]`, that is all HTTP header fields stored in a list of key-value tuples. See RFC 1945 for a listing of all header fields, for example `{date, "Wed, 15 Oct 1997 14:35:17 GMT"}`.

`entity_body` The Entity-Body as defined in RFC 1945, for example data sent from a CGI-script using the POST method.

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, that is `{proceed,NewData} | {break,NewData} | done` which has the following meaning (`OldData` refers to the data field in the incoming `mod` record):

`{proceed,OldData}` Proceed to next module as nothing happened.

`{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 1945.

`{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 EWSAPI Module Sequence does not think otherwise!) including a 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 37].

`{break,NewData}` Has the same semantics as `proceed` above but with one important exception; No more modules in the EWSAPI Module Sequence are executed. Use with care!

`done` No more modules in the EWSAPI 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 EWSAPI 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 EWSAPI 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 35] and `httpd_socket(3)` [page 33].

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 45]. 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 45] which introduces a new context and </Directory> [page 45] 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.

A naive example from mod_log.erl:

```
load([$T,$r,$a,$n,$s,$f,$e,$r,$L,$o,$g,$ |TransferLog],[]) ->
  {ok,[],{transfer_log,httpd_conf:clean(TransferLog)}};
load([$E,$r,$r,$o,$r,$L,$o,$g,$ |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} An invalid directive.

A naive example 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's 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.
```

Keep in mind that there exists numerous utility functions to help you as an EWSAPI module programmer, e.g. nifty lookup of data in ETS-tables/key-value lists and configure utilities. You are well advised to read `httpd_conf(3)` [page 25] and `httpd_util(3)` [page 35].

EWSAPI MODULE INTERACTION

Modules in the EWSAPI Module Sequence [page 29] uses 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 42] 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 44] and `mod_auth:add_user/5` [page 50]. 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 35] and `httpd_conf(3)` [page 25].

BUGS

If a Web browser connect itself to an SSL enabled server using a URL *not* starting with `https://` the server will hang due to an ugly bug in the SSLey package!

SEE ALSO

`httpd_core(3)` [page 27], `httpd_conf(3)` [page 25], `httpd_socket(3)` [page 9], `httpd_util(3)` [page 35], `inets(6)` [page 9], `mod_actions(3)` [page 40], `mod_alias(3)` [page 42], `mod_auth(3)` [page 45], `mod_security(3)` [page 68], `mod_cgi(3)` [page 53], `mod_dir(3)` [page 55], `mod_disk_log(3)` [page 56], `mod_esi(3)` [page 59], `mod_include(3)` [page 63], `mod_log(3)` [page 66]

httpd_conf (Module)

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

Warning:

The current implementation of EWSAPI 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()

- Result = {ok,Directory} | {error,Reason}
- Directory = string()
- Reason = string() | enoent | eaccess | enotdir

`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`.

`is_file(FilePath) -> Result`

Types:

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

`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`.

`make_integer(String) -> Result`

Types:

- String = string()
- Result = {ok,integer()} | {error,nomatch}

`make_integer/1` returns an integer representation of `String`.

SEE ALSO

`httpd(3)` [page 16]

httpd_core (Module)

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

- BindAddress [page 28]
- DefaultType [page 28]
- DocumentRoot [page 29]
- MaxClients [page 29]
- Modules [page 29]
- Port [page 30]
- ServerAdmin [page 30]
- ServerName [page 30]
- ServerRoot [page 31]
- SocketType [page 31]
- SSLCertificateFile [page 31]
- SSLCertificateKeyFile [page 31]
- SSLVerifyClient [page 32]
- KeepAlive [page 32]
- KeepAliveTimeout [page 32]

SECURE SOCKETS LAYER (SSL)

The SSL support is realized using the SSLeay⁹ package. Please refer to `ssl(3)`.

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.

⁹URL: <http://psych.psy.uq.oz.au/~ftp/Crypto/>

- 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 `SSLCertificateFile` [page 31], `SSLCertificateKeyFile` [page 31] and the `SSLVerifyClient` [page 32] 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 31]. Refer to the example server root¹⁰. MIME types may be added as required to the `mime.types` file and the `DefaultType` [page 28] config directive can be used to specify a default mime type.

DIRECTIVE: "BindAddress"

Syntax: `BindAddress address`

Default: `BindAddress *`

Module: `httpd_core(3)` [page 27]

`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. Address can be given either as an IP address or a hostname.

DIRECTIVE: "DefaultType"

Syntax: `DefaultType mime-type`

Default: `- None` - *Module:* `httpd_core(3)` [page 27]

When the server is asked to provide a document type which cannot be determined by the MIME Type Settings [page 28], the server must inform the client about the content type of documents and `mime-type` is used if an unknown type is encountered.

¹⁰In Windows: `%INETS_ROOT%\examples\server_root`. In UNIX: `$INETS_ROOT/examples/server_root`.

DIRECTIVE: "DocumentRoot"

Syntax: DocumentRoot directory-filename

Default: - Mandatory - *Module:* httpd_core(3) [page 27]

DocumentRoot points the Web server to the document space from which to serve documents from. Unless matched by a directive like Alias [page 42], 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: "MaxClients"

Syntax: MaxClients number

Default: MaxClients 150 *Module:* httpd_core(3) [page 27]

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

DIRECTIVE: "Modules"

Syntax: Modules module module ...

Default: Modules mod_get mod_head mod_log

Module: httpd_core(3) [page 27]

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 EWSAPI [page 19] (Erlang Web Server API). 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 [page 23] description in `httpd(3)`.

DIRECTIVE: "Port"

Syntax: Port number

Default: Port 80

Module: httpd_core(3) [page 27]

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 characteristics¹¹ 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 27]

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 27]

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

¹¹In UNIX: /etc/services.

DIRECTIVE: "ServerRoot"

Syntax: ServerRoot directory-filename

Default: - Mandatory -

Module: httpd_core(3) [page 27]

ServerRoot defines a directory-filename where the server has its 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 66]).

DIRECTIVE: "SocketType"

Syntax: SocketType type

Default: SocketType ip_comm

Module: httpd_core(3) [page 27]

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 27] in `httpd(3)`).

DIRECTIVE: "SSLCertificateFile"

Syntax: SSLCertificateFile filename

Default: - None -

Module: httpd_core(3) [page 27]

SSLCertificateFile points at a PEM encoded certificate. Read more about PEM encoded certificates in the SSL application documentation. The dummy certificate `server.pem`¹², 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 27]

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 31].

¹²In 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 27]

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: "KeepAlive"

Syntax: KeepAlive max-requests

Default: - Disabled -

Module: httpd_core(3) [page 27]

This directive enables Keep-Alive support. Set `max-requests` to the maximum number of requests you want the server to serve per connection. A limit is imposed to prevent a client from hogging your server resources. To disable Keep-Alive support, do not set this directive.

The Keep-Alive extension to HTTP, as defined by the HTTP/1.1 draft, allows persistent connections. These long-lived HTTP sessions allow multiple requests to be sent over the same TCP connection, and in some cases have been shown to result in almost 50% speedup in latency times for HTML documents with lots of images.

DIRECTIVE: "KeepAliveTimeout"

Syntax: KeepAliveTimeout seconds

Default: - Disabled -

Module: httpd_core(3) [page 27]

The number of seconds the server will wait for a subsequent request before closing the connection.

SEE ALSO

httpd(3) [page 16]

httpd_socket (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`.

Warning:

The current implementation of EWSAPI is under review and feedback is welcomed.

Exports

`deliver(SocketType,Socket,Binary) -> Result`

Types:

- `SocketType = ip_comm | {ssl,SSLConfigString}`
- `SSLConfigString = string()`
- `Socket = socket()`
- `Binary = binary()`
- `Result = socket_closed | void()`

`deliver/3` sends a `Binary` over a `Socket` in 2kB chunks using the specified `SocketType`. `SSLConfigString` is a SSL configuration string as described in the SSL application documentation.

`peername(SocketType,Socket) -> {Port,IPaddress}`

Types:

- `SocketType = ip_comm | {ssl,SSLConfigString}`
- `SSLConfigString = string()`
- `Socket = socket()`
- `Port = integer()`
- `IPaddress = string()`

`peername/3` returns the `Port` and `IPaddress` of the remote `Socket`. `SSLConfigString` is a SSL configuration string as described in the SSL application documentation.

`resolve() -> HostName`

Types:

- `HostName = string()`

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

SEE ALSO

[httpd\(3\)](#) [page 16]

httpd_util (Module)

This module provides the EWSAPI [page 19] module programmer with miscellaneous utility functions.

Warning:

The current implementation of EWSAPI is under review and feedback is welcomed.

Exports

`decode_base64(Base64String) -> ASCIIStrng`

Types:

- `Base64String = ASCIIStrng = string()`

`decode_base64/1` converts `Base64String` to the plain ascii string (`ASCIIStrng`). The string "BAD!" is returned if `Base64String` is not base64 encoded. Read more about base64 encoding in RFC 1521.

`decode_hex(HexValue) -> DecValue`

Types:

- `HexValue = DecValue = string()`

Converts the hexadecimal value `HexValue` into its decimal equivalent (`DecValue`).

`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(ASCIIStrng) -> Base64String`

Types:

- `ASCIIStrng = string()`
- `Base64String = string()`

`encode_base64` encodes a plain ascii string to a Base64 encoded string. See RFC 1521 for a description of Base64 encoding.

`header(StatusCode,Date)`

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

Types:

- `StatusCode = integer()`
- `Date = rfc1123_date()`
- `MimeType = string()`

`header` returns a HTTP 1.0 header string. The `StatusCode` is one of the status codes defined in RFC 1945 and the `Date` string is RFC 1123 compliant. (See `rfc1123_date/0` [page 38]).

`flatlength(NestedList) -> Size`

Types:

- `NestedList = list()`
- `Size = integer()`

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

`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 directory¹³).

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 directory¹⁴). If no appropriate association can be found the value of DefaultType [page 28] 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 1.0 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 1945.

501 {Method,RequestURI,HTTPVersion}: The HTTP Method, Request-URI and HTTP-Version as defined in RFC 1945.

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".

¹³In Windows: %SERVER_ROOT%\conf\mime.types. In UNIX: \$SERVER_ROOT/conf/mime.types.

¹⁴In Windows: %SERVER_ROOT%\conf\mime.types. In UNIX: \$SERVER_ROOT/conf/mime.types.

`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 = 200 | 201 | 204 | 301 | 302 | 304 | 400 | 401 | 403 | 404 | 500 | 501 | 502 | 504
- Description = `string()`

`reason_phrase` returns the `Description` of an HTTP 1.0 `StatusCode`, for example 200 is "OK" and 201 is "Created". Read RFC 1945 for further information.

`rfc1123_date() -> RFC1123Date`

Types:

- RFC1123Date = `string()`

`rfc1123_date/0` returns the current date in RFC 1123 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 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 1945. 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.

`suffix(FileName) -> 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 (`.`).

`to_lower(String) -> ConvertedString`

Types:

- `String = ConvertedString = string()`

`to_lower/1` converts upper-case letters to lower-case.

`to_upper(String) -> ConvertedString`

Types:

- `String = ConvertedString = string()`

`to_upper/1` converts lower-case letters to upper-case.

SEE ALSO

[httpd\(3\)](#) [page 16]

mod_actions (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 40]
- Script [page 40]

DIRECTIVE: "Action"

Syntax: Action mime-type cgi-script

Default: - None -

Module: mod_actions(3) [page 40]

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_text
Action home-grown/mime-type1 /~bob/do_special_stuff
```

DIRECTIVE: "Script"

Syntax: Script method cgi-script

Default: - None -

Module: mod_actions(3) [page 40]

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 PUT /~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 42].

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 44]

SEE ALSO

`httpd(3)` [page 16], `mod_alias(3)` [page 42]

mod_alias (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 42]
- DirectoryIndex [page 42]
- ScriptAlias [page 43]

DIRECTIVE: "Alias"

Syntax: Alias url-path directory-filename

Default: - None -

Module: mod_alias(3) [page 42]

The Alias directive allows documents to be stored in the local file system instead of the DocumentRoot [page 29] 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 42]

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 42]

The ScriptAlias directive has the same behavior as the Alias [page 42] 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 EWSAPI interaction data, if possible:

{real_name, {Path, AfterPath}} Path and AfterPath is as defined in `httpd_util_split_path/1` [page 39] with one exception - Path has been run through `default_index/2` [page 43].

Uses the following exported EWSAPI functions:

- `mod_alias:default_index/2` [page 43]
- `mod_alias:path/3` [page 44]
- `mod_alias:real_name/3` [page 44]

This module furthermore exports a batch of functions to be used by other EWSAPI 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 42]. 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 `httpd(3)` [page 19].

`path(Data,ConfigDB,RequestURI) -> Path`

Types:

- Data = `interaction_data()`
- ConfigDB = `config_db()`
- RequestURI = Path = `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 43]; Path is returned. If no interaction data has been exported, `ServerRoot` [page 31] is used to generate a file Path. `config_db()` and `interaction_data()` are as defined in `httpd(3)` [page 19].

`real_name(ConfigDB,RequestURI,Aliases) -> Ret`

Types:

- ConfigDB = `config_db()`
- RequestURI = `string()`
- Aliases = `[{FakeName, RealName}]`
- Ret = `{ShortPath, Path, AfterPath}`
- ShortPath = Path = AfterPath = `string()`

`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 39]. Path is generated from ShortPath, that is the result from `default_index/2` [page 43] with ShortPath as an argument. `config_db()` is the server config file in ETS table format as described in `httpd(3)` [page 19].

`real_script_name(ConfigDB,RequestURI,ScriptAliases) -> Ret`

Types:

- ConfigDB = `config_db()`
- RequestURI = `string()`
- ScriptAliases = `[{FakeName, RealName}]`
- Ret = `{ShortPath, AfterPath} | not_a_script`
- ShortPath = AfterPath = `string()`

`real_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 `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 38]. `config_db()` is the server config file in ETS table format as described in `httpd(3)` [page 19].

SEE ALSO

`httpd(3)` [page 16]

mod_auth (Module)

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

- `<Directory>` [page 45]
- `AuthDBType` [page 46]
- `AuthAccessPassword` [page 48]
- `AuthUserFile` [page 46]
- `AuthGroupFile` [page 47]
- `AuthName` [page 48]
- `allow` [page 48]
- `deny` [page 49]
- `require` [page 49]

The `Directory` [page 45] config directive is central to be able to restrict access to certain areas of the server. Please read about the `Directory` [page 45] config directive.

DIRECTIVE: "Directory"

Syntax: `<Directory regexp-filename>`

Default: - None -

Module: `mod_auth(3)` [page 45]

Related: `allow` [page 48], `deny` [page 49], `AuthAccessPassword` [page 48], `AuthGroupFile` [page 47], `AuthUserFile` [page 46], `AuthName` [page 48], `require` [page 49]

`<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 s0mEpAsSw0rD
  AuthDBType plain
  AuthName My Secret Garden
  AuthUserFile /var/tmp/server_root/auth/user
  AuthUserFile /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 45]

Context: <Directory> [page 45]

Related: allow [page 48], deny [page 49], AuthAccessPassword [page 48], AuthName [page 48], AuthUserFile [page 47], AuthUserFile [page 46], require [page 49]

AuthDBType sets the type of authentication database that is used for this directory.

Note:

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

Note:

If you use the `mnesia` storage method, you need to create the `mnesia` tables `httpd_user` and `httpd_group` yourself prior to starting the server.

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 them.

DIRECTIVE: "AuthUserFile"

Syntax: AuthUserFile filename

Default: - None -

Module: mod_auth(3) [page 45]

Context: <Directory> [page 45]

Related: allow [page 48], deny [page 49], AuthDBType [page 46], AuthAccessPassword [page 48], AuthGroupFile [page 47], AuthName [page 48], require [page 49]

`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 50] 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 45]

Context: <Directory> [page 45]

Related: `allow` [page 48], `deny` [page 49], `AuthName` [page 48], `AuthUserFile` [page 46], `AuthDBType` [page 46], `AuthAccessPassword` [page 48], `require` [page 49]

`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 50] 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 45]

Context: <Directory> [page 45]

Related: allow [page 48], deny [page 49], AuthGroupFile [page 47], AuthUserFile [page 46], AuthDBType [page 46], AuthAccessPassword [page 48], require [page 49]

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: - None -

Module: mod_auth(3) [page 45]

Context: <Directory> [page 45]

Related: allow [page 48], deny [page 49], AuthGroupFile [page 47], AuthUserFile [page 46], AuthDBType [page 46], AuthName [page 48], require [page 49]

AuthAccessPassword sets the password required for API calls. All API calls to mod_auth require this password to be specified or they will fail with the error reason `not_authorized`.

DIRECTIVE: "allow"

Syntax: allow from host host ...

Default: allow from all

Module: mod_auth(3) [page 45]

Context: <Directory> [page 45]

Related: AuthAccessPassword [page 48], deny [page 49], AuthGroupFile [page 47], AuthGroupFile [page 46], AuthName [page 48], AuthDBType [page 46] require [page 49]

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 45]

Context: <Directory> [page 45]

Related: allow [page 48], AuthGroupFile [page 47], AuthGroupFile [page 46], AuthName [page 48], AuthDBType [page 46], AuthAccessPassword [page 48], require [page 49]

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 45]

Context: <Directory> [page 45]

Related: allow [page 48], deny [page 49], AuthGroupFile [page 47], AuthUserFile [page 46], AuthName [page 48], AuthDBType [page 46], AuthAccessPassword [page 48]

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 42].

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 44]

Exports

```
add_user(Username, Password, UserData, Port, Dir) -> true | {error, Reason}
```

Types:

- `Username` = `string()`
- `Password` = `string()`
- `UserData` = `term()`
- `Port` = `integer()`
- `Dir` = `string()`
- `Reason` = `term()`

`add_user/5` adds a user to the user database. If the operation is successful, this function returns `true`. If an error occurs, `{error, Reason}` is returned.

```
delete_user(Username, Port, Dir) -> true | {error, Reason}
```

Types:

- `Username` = `string()`
- `Port` = `integer()`
- `Dir` = `string()`
- `Reason` = `term()`

`delete_user/3` deletes a user from the user database. If the operation is successful, this function returns `true`. If an error occurs, `{error, Reason}` is returned.

```
get_user(Username, Port, Dir) -> {ok, #httpd_user} | {error, Reason}
```

Types:

- `Username` = `string()`
- `Port` = `integer()`
- `Dir` = `string()`
- `Reason` = `term()`

`get_user/3` returns a `httpd_user` record containing the userdata for a specific user. If the user cannot be found, `{error, Reason}` is returned.

`list_users(Port, Dir) -> {ok, Users}`

Types:

- `UserName = string()`
- `Port = integer()`
- `Dir = string()`
- `Users = list()`

`list_users/2` returns a list of users in the user database for a specific `Port/Dir`.

`add_group_member(Groupname, Username, Port, Dir) -> true | {error, Reason}`

Types:

- `GroupName = string()`
- `UserName = string()`
- `Port = integer()`
- `Dir = string()`
- `Reason = term()`

`add_group_member/4` 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`

`delete_group_member(Groupname, Username, Port, Dir) -> true | {error, Reason}`

Types:

- `GroupName = string()`
- `UserName = string()`
- `Port = integer()`
- `Dir = string()`
- `Reason = term()`

`delete_group_member/4` adds a user to a group. If the group or the user does not exist, this function returns an error, otherwise it returns `true`.

`list_group_members(Groupname, Port, Dir) -> {ok, Users} | {error, Reason}`

Types:

- `GroupName = string()`
- `Port = integer()`
- `Dir = string()`
- `Users = list()`
- `Reason = term()`

`list_group_members/3` lists the members of a specified group. If the group does not exist or there is an error, `{error, Reason}` is returned.

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

Types:

- `Port = integer()`

- Dir = string()
- Groups = list()
- Reason = term()

list_groups/2 lists all the groups available. If there is an error, {error, Reason} is returned.

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

Types:

- Port = integer()
- Dir = string()
- GroupName = string()
- Reason = term()

delete_group/3 deletes the group specified and returns true. If there is an error, {error, Reason} is returned.

SEE ALSO

httpd(3) [page 16], mod_alias(3) [page 42],

mod_cgi (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 43] config directive is treated as a CGI script. A CGI script is executed by the server and its output is returned to the client.

Support for CGI-1.1 is implemented in accordance with the CGI-1.1 specification¹⁵.

Note:

CGI is currently available for Erlang/OTP running on a UNIX platform. These number of platforms will be increased.

EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:

{new_request_uri, NewRequestURI} as defined in mod_actions(3) [page 41].

{remote_user, RemoteUser} as defined in mod_auth(3) [page 50].

Uses the following EWSAPI functions:

- mod_alias:real_name/3 [page 44]
- mod_alias:real_script_name/3 [page 44]
- mod_cgi:env/3 [page 54]
- mod_cgi:status_code:env/1 [page 54]

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

¹⁵URL: <http://hoohoo.ncsa.uiuc.edu/cgi/>

Exports

`env(Info,Script,AfterScript) -> EnvString`

Types:

- Info = mod_record()
- Script = AfterScript = EnvString = string()

Note:

This function should only be used when implementing CGI-1.1 functionality on UNIX platforms.

`open_port/2` is normally used to start and interact with CGI scripts. `open_port/2` takes an external program as input; `env(1)` (GNU Shell Utility) is typically used in the case of a CGI script. `env(1)` execute the CGI script in a modified environment and takes the CGI script and a string of environment variables as input. `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 CGI-1.1 specification¹⁶. `mod_record()` is a record as defined in the EWSAPI Module Programming [page 19] section of `httpd(3)`.

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

Types:

- CGIOutput = Reason = string()
- StatusCode = integer()

Certain output from CGI scripts has a special meaning, as described in the CGI specification¹⁷, for example if "Location: `http://www.yahoo.com\n\n`" is returned from a CGI script the client gets automatically redirected to Yahoo!¹⁸, using the HTTP 302 status code (RFC 1945).

SEE ALSO

`httpd(3)` [page 16], `mod_auth(3)` [page 45], `mod_security(3)` [page 68], `mod_alias(3)` [page 42], `mod_esi(3)` [page 59], `mod_include(3)` [page 63]

¹⁶URL: <http://hoohoo.ncsa.uiuc.edu/cgi/>

¹⁷URL: <http://hoohoo.ncsa.uiuc.edu/cgi/>

¹⁸URL: <http://www.yahoo.com>

mod_dir (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 29] 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 43].

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 28] section of `httpd_core(3)`.

Uses the following EWSAPI functions:

- `mod_alias:default_index/2` [page 43]
- `mod_alias:path/3` [page 44]

SEE ALSO

`httpd(3)` [page 16], `mod_alias(3)` [page 42]

mod_disk_log (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: "ErrorDiskLog"

Syntax: ErrorDiskLog filename

Default: - None -

Module: mod_disk_log(3) [page 56]

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 31], for example:

```
ErrorDiskLog logs/error_disk_log_8080
```

and errors will be logged in the server root¹⁹ space.

¹⁹In Windows: %SERVER_ROOT%\logs\error_disk_log_8080. In UNIX: \$SERVER_ROOT/logs/error_disk_log_8080.

DIRECTIVE: "ErrorDiskLogSize"

Syntax: ErrorDiskLogSize max-bytes max-files

Default: ErrorDiskLogSize 512000 8

Module: mod_disk_log(3) [page 56]

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: "TransferDiskLog"

Syntax: TransferDiskLog filename

Default: - None -

Module: mod_disk_log(3) [page 56]

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's assumed to be relative to the ServerRoot [page 31], for example:

```
TransferDiskLog logs/transfer_disk_log_8080
```

and errors will be logged in the server root²⁰ space.

DIRECTIVE: "TransferDiskLogSize"

Syntax: TransferDiskLogSize max-bytes max-files

Default: TransferDiskLogSize 512000 8

Module: mod_disk_log(3) [page 56]

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 50].

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

²⁰In Windows: %SERVER_ROOT%\logs\transfer_disk_log_8080. In UNIX: \$SERVER_ROOT/logs/transfer_disk_log_8080.

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 16]. `Date` is a RFC 1123 date string as generated by `httpd_util:rfc1123_date/0` [page 38].

SEE ALSO

`httpd(3)` [page 16], `mod_auth(3)` [page 45], `mod_security(3)` [page 68], `mod_log(3)` [page 66]

mod_esi (Module)

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 53] 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:

- `ErlScripAlias` [page 61]
- `EvalScriptAlias` [page 62]

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, i.e. `Func(Env, Input)`. `Env` contains information about the connecting client (see below), and `Input` the `QueryString` or `PathInfo` as defined in the CGI specification²¹. `***` above depends on how the `ErlScripAlias` [page 61] config directive has been used. Data returned from the function must furthermore take the form as specified in the CGI specification²².

Take a look at `httpd_example.erl` in the code release²³ for a clarifying example. Start an example server as described in `httpd:start/0` [page 18] and test the following from a browser (The server name for your example server *will* differ!):

²¹URL: <http://hoohoo.ncsa.uiuc.edu/cgi/>

²²URL: <http://hoohoo.ncsa.uiuc.edu/cgi/>

²³In Windows: `%INETs\src`. In UNIX: `$INETs/src`.

`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:
[{"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!²⁴ site will promptly be shown in your browser.

Note:

`httpd:parse_query/1` [page 19] 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,-}` or `{-, {socket_closed,-}}`.

²⁴URL: <http://www.yahoo.com>

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.

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 62] config directive has been used. Data returned from the function must furthermore take the form as specified in the CGI specification²⁵.

Take a look at `httpd_example.erl` in the code release²⁶ for an example. Start an example server as described in `httpd:start/0` [page 18] and test the following from a browser (The server name for your example server *will* differ!):

```
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 59]

`ErlScriptAlias` marks all URLs matching `url-path` as `erl` scheme [page 59] 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 59] description above.

²⁵URL: `http://hoohoo.ncsa.uiuc.edu/cgi/`

²⁶In Windows: `%INETS\src`. In UNIX: `$INETS/src`.

DIRECTIVE: "EvalScriptAlias"

Syntax: EvalScriptAlias url-path allowed-module allowed-module ...

Default: - None -

Module: mod_esi(3) [page 59]

EvalScriptAlias marks all URLs matching url-path as eval scheme [page 61] scripts. A matching URL is mapped into a specific module and function. The module must be one of the allowed-module:s. 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 61] description above.

EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:

{remote_user, RemoteUser} as defined in mod_auth(3) [page 50].

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 28] section of httpd_core(3).

Uses the following EWSAPI functions:

- mod_alias:real_name/3 [page 44]
- mod_cgi:status_code/1 [page 54]

SEE ALSO

httpd(3) [page 16], mod_alias(3) [page 42], mod_auth(3) [page 45], mod_security(3) [page 68], mod_cgi(3) [page 53]

mod_include (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 28] 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** or **abbrev. 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 43].
`{remote_user, RemoteUser}` as defined in `mod_auth(3)` [page 50]

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 28] section of `httpd_core(3)`.

Uses the following EWSAPI functions:

- `mod_cgi:env/3` [page 54]
- `mod_alias:path/3` [page 44]
- `mod_alias:real_name/3` [page 44]
- `mod_alias:real_script_name/3` [page 44]

SEE ALSO

`httpd(3)` [page 16], `mod_alias(3)` [page 42], `mod_auth(3)` [page 45], `mod_security(3)` [page 68], `mod_cgi(3)` [page 53]

mod_log (Module)

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 66]

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's assumed to be relative to the ServerRoot [page 31], for example:

```
ErrorLog logs/error_log_8080
```

and errors will be logged in the server root²⁷ space.

²⁷In Windows: %SERVER_ROOT%\logs\error_log_8080. In UNIX: \$SERVER_ROOT/logs/error_log_8080.

DIRECTIVE: "TransferLog"

Syntax: TransferLog filename

Default: - None -

Module: mod_log(3) [page 66]

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's assumed to be relative to the ServerRoot [page 31]. For example:

```
TransferLog logs/access_log_8080
```

and errors will be logged in the server root²⁸ space.

EWSAPI MODULE INTERACTION

Uses the following EWSAPI interaction data, if available:

{remote_user, RemoteUser} as defined in mod_auth(3) [page 50].

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 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 16]. Date is a RFC 1123 date string as generated by httpd_util:rfc1123_date/0 [page 38].

SEE ALSO

httpd(3) [page 16], mod_auth(3) [page 45], mod_security(3) [page 68],
mod_disk_log(3) [page 56]

²⁸In Windows: %SERVER_ROOT%\logs\access_log_8080. In UNIX: \$SERVER_ROOT/logs/access_log_8080.

mod_security (Module)

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 ??]
- SecurityDataFile [page 68]
- SecurityMaxRetries [page 69]
- SecurityBlockTime [page 69]
- SecurityFailExpireTime [page 69]
- SecurityAuthTimeout [page 70]
- SecurityCallbackModule [page 70]

DIRECTIVE: "SecurityDataFile"

Syntax: SecurityDataFile filename

Default: - None -

Module: mod_security(3) [page 68]

Context: <Directory> [page 45]

Related: SecurityMaxRetries [page 69], SecurityBlockTime [page 69], SecurityFailExpireTime [page 69], SecurityAuthTimeout [page 70], SecurityCallbackModule [page 70]

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 68]

Context: <Directory> [page 45]

Related: SecurityDataFile [page 68], SecurityBlockTime [page 69], SecurityFailExpireTime [page 69], SecurityAuthTimeout [page 70], SecurityCallbackModule [page 70]

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 68]

Context: <Directory> [page 45]

Related: SecurityDataFile [page 68], SecurityMaxRetries [page 69], SecurityFailExpireTime [page 69], SecurityAuthTimeout [page 70], SecurityCallbackModule [page 70]

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 68]

Context: <Directory> [page 45]

Related: SecurityDataFile [page 68], SecurityMaxRetries [page 69], SecurityFailExpireTime [page 69], SecurityAuthTimeout [page 70], SecurityCallbackModule [page 70]

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 68]

Context: <Directory> [page 45]

Related: SecurityDataFile [page 68], SecurityMaxRetries [page 69], SecurityFailExpireTime [page 69], SecurityFailExpireTime [page 69], SecurityCallbackModule [page 70]

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 70] function.

DIRECTIVE: "SecurityCallbackModule"

Syntax: SecurityCallbackModule atom()

Default: - None -

Module: mod_security(3) [page 68]

Context: <Directory> [page 45]

Related: SecurityDataFile [page 68], SecurityMaxRetries [page 69], SecurityFailExpireTime [page 69], SecurityFailExpireTime [page 69], SecurityAuthTimeout [page 70]

SecurityCallbackModule specifies the name of a callback module. This module only has one export, event/4 [page 72], which is called whenever a security event occurs. Read the callback module [page 71] documentation to find out more.

Exports

```
list_auth_users(Port) -> Users | []
```

```
list_auth_users(Port, Dir) -> Users | []
```

Types:

- Port = integer()
- Users = list() = [string()]

list_auth_users/1 and list_auth_users/2 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(Port, Dir) -> Users | []
```

Types:

- Port = integer()

- `Users = list() = [string()]`

`list_blocked_users/1` returns a list of users that are currently blocked from access.

`block_user(User, Port, Dir, Seconds) -> true | {error, no_such_directory}`

Types:

- `Port = integer()`
- `User = string()`
- `Dir = string()`
- `Seconds = integer() | infinity`

`block_user/1` blocks the user `User` from the directory `Directory` for a specified amount of time.

`unblock_user(User, Port) -> true | {error, Reason}`

`unblock_user(User, Port, Dir) -> true | {error, Reason}`

Types:

- `Port = integer()`
- `User = string()`
- `Dir = string()`
- `Reason = term()`

`unblock_user/1` 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 72], which is described below.

Exports

`event(What, Port, Dir, Data) -> ignored`

Types:

- `What = atom()`
- `Port = integer()`
- `Dir = string()`
- `What = [Info]`
- `Info = {Name, Value}`

`event/4` is called whenever an event occurs in the `mod_security EWSAPI` module. The `What` argument specifies the type of event that has occurred, and should be one of the following reasons; `auth_fail` (a failed user authentication), `user_block` (a user is being blocked from access) or `user_unblock` (a user is being removed from the block list).

Note:

Note that the `user_unblock` event is not triggered when a user is removed from the block list explicitly using the `unblock_user` function.

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

Modules are typed in *this way*.
Functions are typed in *this way*.

add_group_member/4
 mod_auth , 51

add_user/5
 mod_auth , 50

block_user/4
 mod_security , 71

cd/2
 ftp , 11

check_enum/2
 httpd_conf , 25

clean/1
 httpd_conf , 25

close/1
 ftp , 11

custom_clean/3
 httpd_conf , 25

day/1
 httpd_util , 35

decode_base64/1
 httpd_util , 35

decode_hex/1
 httpd_util , 35

default_index/2
 mod_alias , 43

delete/2
 ftp , 11

delete_group/3
 mod_auth , 52

delete_group_member/4
 mod_auth , 51

delete_user/3
 mod_auth , 50

deliver/3
 httpd_socket , 33

encode_base64/1
 httpd_util , 35

env/3
 mod_cgi , 54

error_log/5
 mod_disk_log , 58
 mod_log , 67

event/4
 mod_security , 71

flatlength/1
 httpd_util , 36

formaterror/1
 ftp , 11

ftp

- cd/2, 11
- close/1, 11
- delete/2, 11
- formaterror/1, 11
- lcd/2, 11
- lpwd/1, 11
- ls/2, 11
- mkdir/2, 12
- nlist/2, 12
- open/2, 12
- pwd/1, 13
- recv/3, 13
- recv_bin/2, 13
- rename/3, 13
- rmdir/2, 13
- send/3, 13
- send_bin/3, 14
- send_chunk/2, 14
- send_chunk_end/1, 14

- send_chunk_start/2, 14
 - type/2, 14
 - user/3, 14
- get_user/3
- mod_auth* , 50
- header/2
- httpd_util* , 36
- header/3
- httpd_util* , 36
- httpd*
- parse_query/1, 19
 - restart/0, 18
 - restart/1, 18
 - start/0, 18
 - start/1, 18
 - stop/0, 18
 - stop/1, 19
- httpd_conf*
- check_enum/2, 25
 - clean/1, 25
 - custom_clean/3, 25
 - is_directory/1, 25
 - is_file/1, 26
 - make_integer/1, 26
- httpd_socket*
- deliver/3, 33
 - peername/2, 33
 - resolve/0, 33
- httpd_util*
- day/1, 35
 - decode_base64/1, 35
 - decode_hex/1, 35
 - encode_base64/1, 35
 - flatlength/1, 36
 - header/2, 36
 - header/3, 36
 - key1search/2, 36
 - key1search/3, 36
 - lookup/2, 36
 - lookup/3, 36
 - lookup_mime/2, 36
 - lookup_mime/3, 36
 - lookup_mime_default/2, 37
 - lookup_mime_default/3, 37
 - message/3, 37
 - month/1, 37
 - multi_lookup/2, 38
 - reason_phrase/1, 38
 - rfc1123_date/0, 38
 - split/3, 38
 - split_path/1, 38
 - split_script_path/1, 38
 - suffix/1, 39
 - to_lower/1, 39
 - to_upper/1, 39
- is_directory/1
- httpd_conf* , 25
- is_file/1
- httpd_conf* , 26
- key1search/2
- httpd_util* , 36
- key1search/3
- httpd_util* , 36
- lcd/2
- ftp* , 11
- list_auth_users/1
- mod_security* , 70
- list_auth_users/2
- mod_security* , 70
- list_blocked_users/1
- mod_security* , 70
- list_blocked_users/2
- mod_security* , 70
- list_group_members/3
- mod_auth* , 51
- list_groups/2
- mod_auth* , 51
- list_users/2
- mod_auth* , 51
- lookup/2
- httpd_util* , 36
- lookup/3
- httpd_util* , 36
- lookup_mime/2
- httpd_util* , 36
- lookup_mime/3
- httpd_util* , 36
- lookup_mime_default/2
- httpd_util* , 37
- lookup_mime_default/3

- httpd_util* , 37
- lpwd/1
 - ftp* , 11
- ls/2
 - ftp* , 11
- make_integer/1
 - httpd_conf* , 26
- message/3
 - httpd_util* , 37
- mkdir/2
 - ftp* , 12
- mod_alias*
 - default_index/2, 43
 - path/3, 44
 - real_name/3, 44
 - real_script_name/3, 44
- mod_auth*
 - add_group_member/4, 51
 - add_user/5, 50
 - delete_group/3, 52
 - delete_group_member/4, 51
 - delete_user/3, 50
 - get_user/3, 50
 - list_group_members/3, 51
 - list_groups/2, 51
 - list_users/2, 51
- mod_cgi*
 - env/3, 54
 - status_code/1, 54
- mod_disk_log*
 - error_log/5, 58
- mod_log*
 - error_log/5, 67
- mod_security*
 - block_user/4, 71
 - event/4, 71
 - list_auth_users/1, 70
 - list_auth_users/2, 70
 - list_blocked_users/1, 70
 - list_blocked_users/2, 70
 - unblock_user/2, 71
 - unblock_user/3, 71
- month/1
 - httpd_util* , 37
- multi_lookup/2
 - httpd_util* , 38
- nlist/2
 - ftp* , 12
- open/2
 - ftp* , 12
- parse_query/1
 - httpd* , 19
- path/3
 - mod_alias* , 44
- peername/2
 - httpd_socket* , 33
- pwd/1
 - ftp* , 13
- real_name/3
 - mod_alias* , 44
- real_script_name/3
 - mod_alias* , 44
- reason_phrase/1
 - httpd_util* , 38
- recv/3
 - ftp* , 13
- recv_bin/2
 - ftp* , 13
- rename/3
 - ftp* , 13
- resolve/0
 - httpd_socket* , 33
- restart/0
 - httpd* , 18
- restart/1
 - httpd* , 18
- rfc1123_date/0
 - httpd_util* , 38
- rmdir/2
 - ftp* , 13
- send/3
 - ftp* , 13
- send_bin/3
 - ftp* , 14
- send_chunk/2
 - ftp* , 14

send_chunk_end/1
 ftp , 14

send_chunk_start/2
 ftp , 14

split/3
 httpd_util , 38

split_path/1
 httpd_util , 38

split_script_path/1
 httpd_util , 38

start/0
 httpd , 18

start/1
 httpd , 18

status_code/1
 mod_cgi , 54

stop/0
 httpd , 18

stop/1
 httpd , 19

suffix/1
 httpd_util , 39

to_lower/1
 httpd_util , 39

to_upper/1
 httpd_util , 39

type/2
 ftp , 14

unblock_user/2
 mod_security , 71

unblock_user/3
 mod_security , 71

user/3
 ftp , 14