Inets

version 4.7

Typeset in LaTeX from SGML source using the DOCBUILDER 3.4 Document System.

Contents

Chapter 1

Inets User's Guide

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

1.1 Introduction

1.1.1 Purpose

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

1.1.2 Prerequisites

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

1.1.3 The Service Concept

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

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

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

For details of exactly what to put in the list of configured services see the documentation for the services that needs configuring.

1.2 FTP Client

1.2.1 Introduction

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

The client supports ipv6 as long as the underlying mechanisms also do so.

1.2.2 Using the FTP Client API

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

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

1.3 HTTP Client

1.3.1 Introduction

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

1.3.2 Configuration

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

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

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

Dir = string()

1.3.3 Using the HTTP Client API

```
1 > application:start(inets).
    ok
```

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

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

An ordinary synchronous request.

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

With all default values, as above, a get request can also be written like this.

```
4 > {ok, {{Version, 200, ReasonPhrase}, Headers, Body}} =
http:request("http://www.erlang.org").
```

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

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

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

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

Send a request with a specified connection header.

```
7 > {ok, {{NewVersion, 200, NewReasonPhrase}, NewHeaders, NewBody}} =
http:request(get, {"http://www.erlang.org", [{"connection", "close"}]},
[], []).
```

1.4 HTTP server

1.4.1 Introduction

The HTTP server, also refered to as httpd, handles HTTP requests as described in RFC 2616 with a few exceptions such as gateway and proxy functionality. (The same is true for servers written by NCSA and others.) The server supports ipv6 as long as the underlying mechanisms also do so.

The server implements numerous features such as SSL (Secure Sockets Layer), ESI (Erlang Scripting Interface), CGI (Common Gateway Interface), User Authentication(using Mnesia, dets or plain text database), Common Logfile Format (with or without disk_log(3) support), URL Aliasing, Action Mappings, Directory Listings and SSI (Server-Side Includes).

The configuration of the server is done using Apache-style configuration directives..

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

1.4.2 Basic 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.config) 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"}]
   }
  ]
 }
].
```

or:

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

According to the new syntax which allows more functionality in the configuration. The possible options here are a customer configurable request accept timeout, the default value is 15000 milliseconds, and some trace functionality to debug the http server. The syntax must match the following grammar:

```
httpd_service() -> {httpd, httpd()}
httpd()         -> [httpd_config()] | file()
httpd_config()  -> {file, file()} |
                   {debug, debug()} |
                   {accept_timeout, integer()}
debug()         -> disable | [debug_options()]
debug_options() -> {all_functions, modules()} |
                   {exported_functions, modules()} |
                   {disable, modules()}
modules()       -> [atom()]
```

{file, file()} corresponds to the functionality of the old version.

{debug, debug()} is the new trace option. It can trace on all functions or only exported functions on choosen modules.

{accept_timeout, integer()} sets the wanted timeout value for the server to set up a request connection.

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

An easy way to test the setup of inets webservers can be done by copying the example server root (UNIX: $INETS_ROOT/examples/server_root/conf/, Windows: %INETS_ROOT%\examples\server_root\conf\) to a specific installation directory (/var/tmp/server_root/conf in this example). Then manualy start the Erlang node, using inets.config.

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

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

Now there should be two HTTP servers started listening on the ports 8888 and 8080. You can test it by using any browser or the inets HTTP client requesting the urls: http://localhost:8888 and http://localhost:8080

1.4.3 Server Runtime Configuration

All functionality in the server can be configured using Apache-style configuration directives stored in a configuration file. A minimal configuration file could look something like:

```
ServerName web.server.net
ServerRoot /var/tmp/server_root
DocumentRoot /var/tmp/server_root/htdocs
```

E.i the syntax is Directive followed by a withspace followed by the value of the directive followed by a new line.

The available directives are described in the section Server Configuration Directives.

1.4.4 Server Configuration Directives

Mandantory Directives

- *DIRECTIVE: "ServerName"*
  Syntax: `ServerName` fully-qualified domain name
  Default: - Mandatory -
  `ServerName` sets the `fully-qualified domain name` of the server.
- *DIRECTIVE: "ServerRoot"*
  Syntax: `ServerRoot` directory-filename
  Default: - Mandatory -
  `ServerRoot` defines a `directory-filename` where the server has it's operational home, e.g. used to store log files and system icons. Relative paths specified in the config file refer to this `directory-filename`.
- *DIRECTIVE: "DocumentRoot"*
  Syntax: `DocumentRoot` directory-filename
  Default: - Mandatory -
  `DocumentRoot` points the Web server to the document space from which to serve documents from. Unless matched by a directive like Alias, 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`.

Communication Directives

- *DIRECTIVE: "BindAddress"*
  Syntax: `BindAddress` address
  Default: `BindAddress *`
  `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: "Port"*
  Syntax: `Port` number
  Default: `Port 80`
  `Port` defines which port `number` the server should use (0 to 65535). Certain port numbers are reserved for particular protocols, i.e. examine your OS characteristics[1] 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: "SocketType"*
  Syntax: `SocketType` type
  Default: `SocketType ip_comm`
  `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.

[1] In UNIX: `/etc/services`.

Limit Directives

- *DIRECTIVE: "DisableChunkedTransferEncodingSend"*
  Syntax: `DisableChunkedTransferEncodingSend` true | false
  Default: false
  This directive tells the server whether to use chunked transfer-encoding when sending a response to a HTTP/1.1 client.
- *DIRECTIVE: "KeepAlive"*
  Syntax: `KeepAlive` true | false
  Default: true
  This directive tells the server whether to use persistent connection or not when the client claims to be HTTP/1.1 compliant.*Note:*the value of KeepAlive has changed from previous versions to be compliant with Apache.
- *DIRECTIVE: "KeepAliveTimeout"*
  Syntax: `KeepAliveTimeout` seconds
  *Default:*150
  The number of seconds the server will wait for a subsequent request from the client before closing the connection. If the load on the server is high you may want to shorten this.
- *DIRECTIVE: "MaxBodyAction"*
  Syntax: `MaxBodyAction` action
  Default: `MaxBodyAction close`
  `MaxBodyAction` specifies the action to be taken when the message body limit has been passed.

  `close` the default and preferred communication type. ip_comm is also used for all remote message passing in Erlang.

  `reply414` a reply (status) message with code 414 will be sent to the client *prior* to closing the socket. Note that this code is *not* defined in the HTTP/1.0 version of the protocol.
- *DIRECTIVE: "MaxBodySize"*
  Syntax: `MaxBodySize` size
  Default: `MaxBodySize nolimit`
  `MaxBodySize` limits the `size` of the message body of HTTP request. The reply to this is specified by the `MaxBodyAction` directive. Valid size is:

  `nolimit` the default message body limit, e.g. no limit.

  `integer()` any positive number.
- *DIRECTIVE: "MaxClients"*
  Syntax: `MaxClients` number
  Default: `MaxClients 150`
  `MaxClients` limits the `number` of simultaneous requests that can be supported. No more than this `number` of child server process's can be created.
- *DIRECTIVE: "MaxHeaderAction"*
  Syntax: `MaxHeaderAction` action
  Default: `MaxHeaderAction close`
  `MaxHeaderAction` specifies the action to be taken when the message Header limit has been passed.

  `close` the socket is closed without any message to the client. This is the default action.

  `reply414` a reply (status) message with code 414 will be sent to the client *prior* to closing the socket. Note that this code is *not* defined in the HTTP/1.0 version of the protocol.
- *DIRECTIVE: "MaxHeaderSize"*
  Syntax: `MaxHeaderSize` size

Default: `MaxHeaderSize 10240`
`MaxHeaderSize` limits the `size` of the message header of HTTP request. The reply to this is specified by the `MaxHeaderAction` directive. Valid size is:

`integer()` any positive number (default is 10240)

`nolimit` no limit should be applied

- *DIRECTIVE: "MaxKeepAliveRequests"*
Syntax: `MaxKeepAliveRequests` NumberOfRequests
Default:- Disabled -
The number of request that a client can do on one connection. When the server has responded to the number of requests defined by MaxKeepAliveRequests the server close the connection. The server will close it even if there are queued request.

Administrative Directives

- *DIRECTIVE: "DefaultType"*
Syntax: `DefaultType` mime-type
Default: - None -
When the server is asked to provide a document type which cannot be determined by the MIME Type Settings, the server must inform the client about the content type of documents and `mime-type` is used if an unknown type is encountered.
- *DIRECTIVE: "Modules"*
Syntax: `Modules` module module ...
Default: `Modules mod_get mod_head mod_log`
`Modules` defines which Erlang Webserver API 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 section Erlang Web Server API. The server executes functionality in each module, from left to right (from now on called *Erlang Webserver API Module Sequence*).
Before altering the Erlang Webserver API Modules Sequence please observe what types of data each module uses and propagates.
- *DIRECTIVE: "ServerAdmin"*
Syntax: `ServerAdmin` email-address
Default: `ServerAdmin unknown@unknown`
`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
```

SSL Directives

- *DIRECTIVE: "SSLCACertificateFile"*
Syntax: `SSLCACertificateFile` filename
Default: - None -
`SSLCACertificateFile` points at a PEM encoded certificate of the certification authorities. Read more about PEM encoded certificates in the SSL application documentation. Read more about PEM encoded certificates in the SSL application documentation.
- *DIRECTIVE: "SSLCertificateFile"*
Syntax: `SSLCertificateFile` filename

Default: - None -
`SSLCertificateFile` points at a PEM encoded certificate. Read more about PEM encoded certificates in the SSL application documentation. The dummy certificate server.pem[2], 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 -
  `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 .
- *DIRECTIVE: "SSLVerifyClient"*
  Syntax: `SSLVerifyClient` type
  Default: - None -
  Set `type` to:

  0 if no client certificate is required.

  1 if the client *may* present a valid certificate.

  2 if the client *must* present a valid certificate.

  3 if the client *may* present a valid certificate but it is *not* required to have a valid CA.

  Read more about SSL in the application documentation.
- *DIRECTIVE: "SSLVerifyDepth"*
  Syntax: `SSLVerifyDepth` integer
  Default: - None -
  This directive specifies how far up or down the (certification) chain we are prepared to go before giving up.
  Read more about SSL in the application documentation.
- *DIRECTIVE: "SSLCiphers"*
  Syntax: `SSLCiphers` ciphers
  Default: - None -
  `SSLCihers` is a colon separated list of ciphers.
  Read more about SSL in the application documentation.
- *DIRECTIVE: "SSLPasswordCallbackFunction"*
  Syntax: `SSLPasswordCallbackFunction` function
  Default: - None -
  The `SSLPasswordCallbackFunction` function in module `SSLPasswordCallbackModule` is called in order to retrieve the user's password.
  Read more about SSL in the application documentation.
- *DIRECTIVE: "SSLPasswordCallbackModule" Syntax:* `SSLPasswordCallbackModule` function
  Default: - None -
  The `SSLPasswordCallbackFunction` function in the `SSLPasswordCallbackModule` module is called in order to retrieve the user's password.
  Read more about SSL in the application documentation.

URL Aliasing

- *DIRECTIVE: "Alias"*
  Syntax: `Alias` url-path directory-filename

[2] In Windows: `%INETS%\examples\server_root\ssl\`. In UNIX: `$INETS/examples/server_root/ssl/`.

Default: - None -
The Alias directive allows documents to be stored in the local file system instead of the DocumentRoot 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 -
  `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 -
  The ScriptAlias directive has the same behavior as the Alias 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`.

CGI Directives

- *DIRECTIVE: "ScriptNoCache"*
  Syntax: `ScritpNoCache` true | false
  Default: - false -
  If ScriptNoCache is set to true the Web server will by default add the header fields necessary to prevent proxies from caching the page. Generally this is something you want.

```
ScriptNoCache true
```

- *DIRECTIVE: "ScriptTimeout"*
  Syntax: `ScritpTimeout` Seconds
  Default: 15
  The time in seconds the web server will wait between each chunk of data from the script. If the CGI-script not delivers any data before the timeout the connection to the client will be closed.

```
ScriptTimeout  15
```

- *DIRECTIVE: "Action"*
  Syntax: `Action` mime-type cgi-script
  Default: - None -
  `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 -
  `Script` adds an action, which will activate a `cgi-script` whenever a file is requested using a certain HTTP `method`. The `method` is either `GET` or `POST` as defined in RFC 1945. It propagates the URL and file path of the requested document using the standard CGI `PATH_INFO` and `PATH_TRANSLATED` environment variables.
  Examples:

```
Script GET /cgi-bin/get
Script POST /~bob/put_and_a_little_more
```

ESI Directives

- *DIRECTIVE: "ErlScriptAlias"*
  Syntax: `ErlScriptAlias` url-path allowed-module allowed-module ...
  Default: - None -
  `ErlScriptAlias` marks all URLs matching `url-path` as erl scheme 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`.
- *DIRECTIVE: "ErlScriptNoCache"*
  Syntax: `ErlScriptNoCache` true | false
  Default: false
  If `ErlScriptNoCache` is set to true the server will add http header fields that prevents proxies from caching the page. This is generally a good idea for dynamic content, since the content often vary between each request.

```
ErlScriptNoCache true
```

- *DIRECTIVE: "ErlScriptTimeout"*
  Syntax: ErlScriptTimeout seconds
  Default: 15
  If ErlScriptTimeout sets the time in seconds the server will wait between each chunk of data is delivered through mod_esi:deliver/2 when the new Erl Scheme format, that takes three argument is used.

```
ErlScriptTimeout 15
```

- *DIRECTIVE: "EvalScriptAlias"*
  Syntax: EvalScriptAlias url-path allowed-module allowed-module ...
  Default: - None -
  EvalScriptAlias marks all URLs matching url-path as eval scheme 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.

Auth Directives

- *DIRECTIVE: "Directory"*
  Syntax: <Directory regexp-filename>
  Default: - None -
  <Directory> and </Directory> are used to enclose a group of directives which applies only to the named directory and sub-directories of that directory. regexp-filename is an extended regular expression (See regexp(3)). For example:

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

  If multiple directory sections match the directory (or its parents), then the directives are applied with the shortest match first. For example if you have one directory section for garden/ and one for garden/flowers, the garden/ section matches first.
- *DIRECTIVE: "AuthDBType"*
  Syntax: AuthDBType plain | dets | mnesia
  Default: - None -
  Context: Directory

AuthDBType sets the type of authentication database that is used for the directory.The key difference between the different methods is that dynamic data can be saved when Mnesia and Dets is used.
If Mnesia is used as storage method, Mnesia must be started prio to the webserver. The first time Mnesia is started the schema and the tables must be created before Mnesia is started. A naive example of a module with two functions that creates and start mnesia is provided here. The function shall be sued the first time. first_start/0 creates the schema and the tables. The second function start/0 shall be used in consecutive startups. start/0 Starts Mnesia and wait for the tables to be initiated. This function must only be used when the schema and the tables already is created.

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

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

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

To create the Mnesia tables we use two records defined in mod_auth.hrl so the file must be included.
The first function first_start/0 creates a schema that specify on which nodes the database shall reside. Then it starts Mnesia and creates the tables. The first argument is the name of the tables, the second argument is a list of options how the table will be created, see Mnesia documentation for more information. Since the current implementation of the mod_auth_mnesia saves one row for each user the type must be bag.
When the schema and the tables is created the second function start/0shall be used to start Mensia. It starts Mnesia and wait for the tables to be loaded. Mnesia use the directory specified as mnesia_dir at startup if specified, otherwise Mnesia use the current directory.
For security reasons, make sure that the Mnesia tables are stored outside the document tree of the Web server. If it is placed in the directory which it protects, clients will be able to download the tables.
Only the dets and mnesia storage methods allow writing of dynamic user data to disk. plain is a read only method.

- *DIRECTIVE: "AuthUserFile"*
  Syntax: AuthUserFile filename
  Default: - None -
  Context: Directory
  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 functions in mod_auth module to create / edit the user database.
This directive is ignored if using the `mnesia` storage method.
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 -
Context: Directory
`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 for mod_auth module to create / edit the group database.
This directive is ignored if using the `mnesia` storage method.
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 -
Context: Directory
`AuthName` sets the name of the authorization realm (`auth-domain`) for a directory. This string informs the client about which user name and password to use.

- *DIRECTIVE: "AuthAccessPassword"*
Syntax: `AuthAccessPassword` password
Default: NoPassword
Context: Directory
If `AuthAccessPassword` is set to other than NoPassword the password is required for all API calls. If the password is set to DummyPassword the password must be changed before any other API calls. To secure the authenticating data the password must be changed after the webserver is started since it otherwise is written in clear text in the configuration file.

- *DIRECTIVE: "allow"*
Syntax: `allow from` host host ...
Default: `allow from all`
Context: Directory
`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`
  Context: Directory
  `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 -
  Context: Directory
  `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.

Htacess Authentication Directives

- *DIRECTIVE: "AccessFileName" Syntax:* `AccessFileNameFileName1 FileName2`
  Default: .htaccess
  `AccessFileName` Specify which filenames that are used for access-files. When a request comes every directory in the path to the requested asset will be searched after files with the names specified by this parameter. If such a file is found the file will be parsed and the restrictions specified in it will be applied to the request.

Auth Filter Directives

- *DIRECTIVE: "SecurityDataFile"*
  Syntax: `SecurityDataFile` filename
  Default: - None -
  Context: Directory

`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.
Several directories can have the same `SecurityDataFile`.

- *DIRECTIVE: "SecurityMaxRetries"*
  Syntax: `SecurityMaxRetries` integer() | infinity
  Default: 3
  Context:
  `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.
  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
  Context: Directory
  `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
  Context: Directory
  `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
  Context: Directory
  `SecurityAuthTimeout` specifies the number of seconds a successful user authentication is remembered. After this time has passed, the authentication will no longer be reported.
- *DIRECTIVE: "SecurityCallbackModule"*
  Syntax: `SecurityCallbackModule` atom()
  Default: - None -
  Context: Directory
  `SecurityCallbackModule` specifies the name of a callback module.

Logging Directives

- *DIRECTIVE: "ErrorLog"*
  Syntax: `ErrorLog` filename
  Default: - None -
  `ErrorLog` defines the `filename` of the error log file to be used to log server errors. If the `filename` does not begin with a slash (/) it is assumed to be relative to the ServerRoot, for example:

```
ErrorLog logs/error_log_8080
```

  and errors will be logged in the server root[3] space.

[3] In Windows: `%SERVER_ROOT%\logs\error_log_8080`. In UNIX: `$SERVER_ROOT/logs/error_log_8080`.

- *DIRECTIVE: "SecurityLog"*
  Syntax: SecurityLog filename
  Default: - None -
  SecurityLog defines the filename of the access log file to be used to log security events. If the filename does not begin with a slash (/) it is assumed to be relative to the ServerRoot. For example:

```
SecurityLog logs/security_log_8080
```

  and security events will be logged in the server root[4] space.
- *DIRECTIVE: "TransferLog"*
  Syntax: TransferLog filename
  Default: - None -
  TransferLog defines the filename of the access log file to be used to log incoming requests. If the filename does not begin with a slash (/) it is assumed to be relative to the ServerRoot. For example:

```
TransferLog logs/access_log_8080
```

  and errors will be logged in the server root[5] space.

Disk Log Directives

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

```
DiskLogFormat external
```

- *DIRECTIVE: "ErrorDiskLog"*
  Syntax: ErrorDiskLog filename
  Default: - None -
  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, for example:

```
ErrorDiskLog logs/error_disk_log_8080
```

  and errors will be logged in the server root[6] space.
- *DIRECTIVE: "ErrorDiskLogSize"*
  Syntax: ErrorDiskLogSize max-bytes max-files

[4] In Windows: %SERVER_ROOT%\logs\security_log_8080. In UNIX: $SERVER_ROOT/logs/security_log_8080.
[5] In Windows: %SERVER_ROOT%\logs\access_log_8080. In UNIX: $SERVER_ROOT/logs/access_log_8080.
[6] In Windows: %SERVER_ROOT%\logs\error_disk_log_8080. In UNIX: $SERVER_ROOT/logs/error_disk_log_8080.

Default: `ErrorDiskLogSize 512000 8`
`ErrorDiskLogSize` defines the properties of the `(disk_log(3))` error log file. The `disk_log(3)` error log file is of type *wrap log* and `max-bytes` will be written to each file and `max-files` will be used before the first file is truncated and reused.

- *DIRECTIVE: "SecurityDiskLog"*
  Syntax: `SecurityDiskLog` filename
  Default: - None -
  `SecurityDiskLog` defines the `filename` of the `(disk_log(3))` access log file which logs incoming security events i.e authenticated requests. If the `filename` does not begin with a slash (`/`) it is assumed to be relative to the ServerRoot.
- *DIRECTIVE: "SecurityDiskLogSize"*
  Syntax: `SecurityDiskLogSize` max-bytes max-files
  Default: `SecurityDiskLogSize 512000 8`
  `SecurityDiskLogSize` defines the properties of the `disk_log(3)` access log file. The `disk_log(3)` access log file is of type *wrap log* and `max-bytes` will be written to each file and `max-files` will be used before the first file is truncated and reused.
- *DIRECTIVE: "TransferDiskLog"*
  Syntax: `TransferDiskLog` filename
  Default: - None -
  `TransferDiskLog` defines the `filename` of the `(disk_log(3))` access log file which logs incoming requests. If the `filename` does not begin with a slash (`/`) it is assumed to be relative to the ServerRoot, for example:

```
TransferDiskLog logs/transfer_disk_log_8080
```

  and errors will be logged in the server root[7] space.
- *DIRECTIVE: "TransferDiskLogSize"*
  Syntax: `TransferDiskLogSize` max-bytes max-files
  Default: `TransferDiskLogSize 512000 8`
  `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.

1.4.5 Mime Type Configuration

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. MIME types may be added as required to the mime.types file and the DefaultType config directive can be used to specify a default mime type. An example of a very small mime.types file:

```
# MIME type                     Extension
text/html                   html htm
text/plain                  asc txt
```

[7] In Windows: `%SERVER_ROOT%\logs\transfer_disk_log_8080`. In UNIX: `$SERVER_ROOT/logs/transfer_disk_log_8080`.

1.4.6 Htaccess - User Configurable Authentication.

If users of the webserver needs to manage authentication of webpages that are local to their user and do not have server administrative privileges. They can use the per-directory runtime configurable user-authentication scheme that Inets calls htaccess. It works the following way:

- Each directory in the path to the requested asset is searched for an access-file (default .htaccess), that restricts the webservers rights to respond to a request. If an access-file is found the rules in that file is applied to the request.
- The rules in an access-file applies both to files in the same directories and in subdirectories. If there exists more than one access-file in the path to an asset, the rules in the access-file nearest the requested asset will be applied.
- To change the rules that restricts the use of an asset. The user only needs to have write access to the directory where the asset exists.
- All the access-files in the path to a requested asset is read once per request, this means that the load on the server will increase when this scheme is used.
- If a directory is limited both by auth directives in the HTTP server configuration file and by the htaccess files. The user must be allowed to get access the file by both methods for the request to succeed.

Access Files Directives

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

- *DIRECTIVE: "allow" Syntax:* `Allow` from subnet subnet|from all
  Default: `from all`
  Same as the directive allow for the server config file.
- *DIRECTIVE: "AllowOverRide" Syntax:* `AllowOverRide` all | none | Directives
  Default: `- None -`
  `AllowOverRide` Specify which parameters that not access-files in subdirectories are allowed to alter the value for. If the parameter is set to none no more access-files will be parsed.
  If only one access-file exists setting this parameter to none can lessen the burden on the server since the server will stop looking for access-files.
- *DIRECTIVE: "AuthGroupfile" Syntax:* `AuthGroupFile` Filename
  Default: `- None -`
  AuthGroupFile indicates which file that contains the list of groups. Filename must contain the absolute path to the file. The format of the file is one group per row and every row contains the name of the group and the members of the group separated by a space, for example:

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

- *DIRECTIVE: "AuthName" Syntax:* `AuthName` auth-domain
  Default: `- None -`
  Same as the directive AuthName for the server config file.

- *DIRECTIVE: "AuthType" Syntax:* `AuthType` Basic
  Default: `Basic`
  `AuthType` Specify which authentication scheme that shall be used. Today only Basic Authenticating using UUEncoding of the password and user ID is implemented.
- *DIRECTIVE: "AuthUserFile" Syntax:* `AuthUserFile` Filename
  Default: `- None -`
  `AuthUserFile` indicate which file that contains the list of users. Filename must contain the absolute path to the file. The users name and password are not encrypted so do not place the file with users in a directory that is accessible via the webserver. The format of the file is one user per row and every row contains User Name and Password separated by a colon, for example:

```
UserName:Password
UserName:Password
```

- *DIRECTIVE: "deny" Syntax:* `deny` from subnet subnet|from all
  Context: Limit
  Same as the directive deny for the server config file.
- *DIRECTIVE: "Limit"*
  Syntax: <`Limit` RequestMethods>
  Default: - None -
  <`Limit`> and </Limit> are used to enclose a group of directives which applies only to requests using the specified methods. If no request method is specified all request methods are verified against the restrictions.

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

- *DIRECTIVE: "order"*
  Syntax: `order` allow deny | deny allow
  Default: allow deny
  `order`, defines if the deny or allow control shall be preformed first.
  If the order is set to allow deny, then first the users network address is controlled to be in the allow subset. If the users network address is not in the allowed subset he will be denied to get the asset. If the network-address is in the allowed subset then a second control will be preformed, that the users network address is not in the subset of network addresses that shall be denied as specified by the deny parameter.
  If the order is set to deny allow then only users from networks specified to be in the allowed subset will succeed to request assets in the limited area.
- *DIRECTIVE: "require" Syntax:* `require` group group1 group2...|user user1 user2...
  Default: `- None -`
  Context: Limit
  See the require directive in the documentation of mod_auth(3) for more information.

1.4.7 Dynamic Web Pages

The Inets HTTP server provides two ways of creating dynamic web pages, each with its own advantages and disadvantages.

First there are CGI-scripts that can be written in any programming language. CGI-scripts are standardized and supported by most webservers. The drawback with CGI-scripts is that they are resource intensive because of their design. CGI requires the server to fork a new OS process for each executable it needs to start.

Second there are ESI-functions that provide a tight and efficient interface to the execution of Erlang functions, this interface on the other hand is Inets specific.

The Common Gateway Interface (CGI) Version 1.1, RFC 3875.

The mod_cgi module makes it possible to execute CGI scripts in the server. A file that matches the definition of a ScriptAlias config directive is treated as a CGI script. A CGI script is executed by the server and it's output is returned to the client.

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

```
"Content-Type:text/plain\nAccept-Ranges:none\n\nsome very
        plain text"
```

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

Support for CGI-1.1 is implemented in accordance with the RFC 3875.

Erlang Server Interface (ESI)

The erlang server interface is implemented by the module mod_esi.

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/***/Module[:/]Function(?QueryString|/PathInfo)
```

*** above depends on how the ErlScriptAlias config directive has been used

The module (Module) referred to must be found in the code path, and it must define a function (Function) with an arity of two or three. It is preferable to implement a funtion with arity three as it permitts you to send chunks of the webpage beeing generated to the client during the generation phase instead of first generating the whole web page and then sending it to the client. The option to implement a function with arity two is only keept for backwardcompatibilty reasons. See mod_esi(3) [page 78] for implementation details of the esi callback function.

EVAL Scheme 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)
```

*** above depends on how the ErlScriptAlias config directive has been used

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. Data returned from the function must furthermore take the form as specified in the CGI specification. See mod_esi(3) [page 78] for implementation details of the esi callback function.

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

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

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

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

1.4.8 Logging

There are three types of logs supported. Transfer logs, security logs and error logs. The de-facto standard Common Logfile Format is used for the transfer and security logging. 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.

Internal server errors are recorde in the error log file. The format of this file is a more ad hoc format than the logs using Common Logfile Format, but conforms to the following syntax:

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

1.4.9 Server Side Includes

Server Side Includes enables the server to run code embedded in HTML pages to generate the response to the client.

> **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, 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.

1.4.10 The Erlang Webserver API

The process of handling a HTTP request involves several steps such as:

- Seting up connections, sending and receiving data.
- URI to filename translation
- Authenication/access cheks.
- Retriving/generating the response.
- Logging

To provide customization and extensibility of the HTTP servers request handling most of these steps are handled by one or more modules that may be replaced or removed at runtime, and ofcourse new ones can be added. For each request all modules will be traversed in the order specified by the modules directive in the server configuration file. Some parts mainly the communication related steps are considered server core functionallity and are not implemented using the Erlang Webserver API. A description of functionality implemented by the Erlang Webserver API is described in the section Inets Webserver Modules.

A module can use data generated by previous modules in the Erlang Webserver API module sequence or generate data to be used by consecutive Erlang Webserver API modules. This is made possible due to an internal list of key-value tuples, also refered to as interaction data.

> Note:
> Interaction data enforces module dependencies and should be avoided if possible. This means the order of modules in the Modules config directive is significant.

API Description

Each module implements server functionality using the Erlang Webserver API should implement the following call back functions:

- do/1 (mandatory) - the function called when a request should be handled.
- load/2
- store/2
- remove/1

The latter functions are needed only when new config directives are to be introduced. For details see httpd(3) [page 56]

1.4.11 Inets Webserver Modules

The convention is that all modules implementing some webserver functionallity has the name mod_*. When configuring the webserver an appropriate selection of these modules should be present in the Module directve. Please note that there are some interaction dependencies to take into account so the order of the modules can not be totally random.

mod_action - Filetype/Method-Based Script Execution.

Runs CGI scripts whenever a file of a certain type or HTTP method (See RFC 1945) is requested.

Uses the following Erlang Webserver API interaction data, if available:

- real_name - from mod_alias

Exports the following Erlang Webserver API interaction data, if possible:

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

mod_alias - URL Aliasing

This module makes it possible to map different parts of the host file system into the document tree e.i. creates aliases and redirections.

Exports the following Erlang Webserver API interaction data, if possible:

`{real_name, PathData}` PathData is the argument used for API function mod_alias:path/3.

mod_auth - User Authentication

This module provides for basic user authentication using textual files, dets databases as well as mnesia databases.

Uses the following Erlang Webserver API interaction data, if available:

- real_name - from mod_alias

Exports the following Erlang Webserver API interaction data, if possible:

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

mod_cgi - CGI Scripts

This module handles invoking of CGI scripts

mod_dir - Directories

This module generates an HTML directory listing (Apache-style) if a client sends a request for a directory instead of a file. This module needs to be removed from the Modules config directive if directory listings is unwanted.

Uses the following Erlang Webserver API interaction data, if available:

- real_name - from mod_alias

Exports the following Erlang Webserver API interaction data, if possible:

`{mime_type, MimeType}` The file suffix of the incoming URL mapped into a `MimeType`.

mod_disk_log - Logging Using disk_log.

Standard logging using the "Common Logfile Format" and disk_log(3).

Uses the following Erlang Webserver API interaction data, if available:

- remote_user - from mod_auth

mod_esi - Erlang Server Interface

This module implements the Erlang Server Interface (ESI) that provides a tight and efficient interface to the execution of Erlang functions.

Uses the following Erlang Webserver API interaction data, if available:

- remote_user - from mod_auth

Exports the following Erlang Webserver API interaction data, if possible:

`{mime_type, MimeType}` The file suffix of the incoming URL mapped into a `MimeType`

mod_get - Regular GET Requests

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

Uses the following Erlang Webserver API interaction data, if available:

- real_name - from mod_alias

mod_head - Regular HEAD Requests

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

Uses the following Erlang Webserver API interaction data, if available:

- real_name - from mod_alias

mod_htacess - User Configurable Access

This module provides per-directory user configurable access control.

Uses the following Erlang Webserver API interaction data, if available:

- real_name - from mod_alias

Exports the following Erlang Webserver API interaction data, if possible:

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

mod_include - SSI

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).

Uses the following Erlang Webserver API interaction data, if available:

- real_name - from mod_alias
- remote_user - from mod_auth

Exports the following Erlang Webserver API 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 section.

mod_log - Logging Using Text Files.

Standard logging using the “Common Logfile Format” and text files.

Uses the following Erlang Webserver API interaction data, if available:

- remote_user - from mod_auth

mod_range - Requests with Range Headers

This module response to requests for one or many ranges of a file. This is especially useful when downloading large files, since a broken download may be resumed.

Note that request for multiple parts of a document will report a size of zero to the log file.

Uses the following Erlang Webserver API interaction data, if available:

- real_name - from mod_alias

mod_response_control - Requests with If* Headers

This module controls that the conditions in the requests is fullfilled. For example a request may specify that the answer only is of interest if the content is unchanged since last retrieval. Or if the content is changed the range-request shall be converted to a request for the whole file instead.

If a client sends more then one of the header fields that restricts the servers right to respond, the standard does not specify how this shall be handled. httpd will control each field in the following order and if one of the fields not match the current state the request will be rejected with a proper response.
1.If-modified
2.If-Unmodified
3.If-Match
4.If-Nomatch
Uses the following Erlang Webserver API interaction data, if available:

- real_name - from mod_alias

Exports the following Erlang Webserver API interaction data, if possible:

`{if_range, send_file}` The conditions for the range request was not fullfilled. The response must not be treated as a range request, instead it must be treated as a ordinary get request.

mod_security - Security Filter

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.

mod_trace - TRACE Request

mod_trace is responsible for handling of TRACE requests. Trace is a new request method in HTTP/1.1. The intended use of trace requests is for testing. The body of the trace response is the request message that the responding Web server or proxy received.

Inets Reference Manual

Short Summaries

- Erlang Module ftp [page 40] – A File Transfer Protocol client
- Erlang Module http [page 50] – An HTTP/1.1 client
- Erlang Module http_base_64 [page 55] – Implements base 64 encode and decode, see RFC2045.
- Erlang Module httpd [page 56] – An implementation of an HTTP 1.1 compliant Web server, as defined in RFC 2616.
- Erlang Module httpd_conf [page 62] – Configuration utility functions to be used by the Erlang Webserver API programmer.
- Erlang Module httpd_socket [page 64] – Communication utility functions to be used by the Erlang Webserver API programmer.
- Erlang Module httpd_util [page 65] – Miscellaneous utility functions to be used when implementing Erlang Webserver API modules.
- Erlang Module mod_alias [page 71] – URL ailasing.
- Erlang Module mod_auth [page 73] – User authentication using text files, dets or mnesia database.
- Erlang Module mod_esi [page 78] – Erlang Server Interface
- Erlang Module mod_security [page 80] – Security Audit and Trailing Functionality
- Erlang Module tftp [page 83] – Trivial FTP

ftp

The following functions are exported:

- `account(Pid, Account) -> ok | {error, Reason}`
  [page 41] Specify which account to use.
- `append(Pid, LocalFile [, RemoteFile]) -> ok | {error, Reason}`
  [page 41] Transfer file to remote server, and append it to Remotefile.
- `append_bin(Pid, Bin, RemoteFile) -> ok | {error, Reason}`
  [page 41] Transfer a binary into a remote file.
- `append_chunk(Pid, Bin) -> ok | {error, Reason}`
  [page 41] append a chunk to the remote file.
- `append_chunk_start(Pid, File) -> ok | {error, Reason}`
  [page 41] Start transfer of file chunks for appending to File.

- `append_chunk_end(Pid) -> ok | {error, Reason}`
  [page 42] Stop transfer of chunks for appending.
- `cd(Pid, Dir) -> ok | {error, Reason}`
  [page 42] Change remote working directory.
- `close(Pid) -> ok`
  [page 42] End ftp session.
- `delete(Pid, File) -> ok | {error, Reason}`
  [page 42] Delete a file at the remote server..
- `formaterror(Tag) -> string()`
  [page 42] Return error diagnostics.
- `lcd(Pid, Dir) -> ok | {error, Reason}`
  [page 42] Change local working directory.
- `lpwd(Pid) -> {ok, Dir}`
  [page 42] Get local current working directory.
- `ls(Pid [, Dir]) -> {ok, Listing} | {error, Reason}`
  [page 43] List contents of remote directory.
- `mkdir(Pid, Dir) -> ok | {error, Reason}`
  [page 43] Create remote directory.
- `nlist(Pid [, Dir]) -> {ok, Listing} | {error, Reason}`
  [page 43] List contents of remote directory.
- `open(Host [, Port] [, Flags]) -> {ok, Pid} | {error, Reason}`
  [page 43] Start an ftp client.
- `open({option_list, Option_list}) -> {ok, Pid} | {error, Reason}`
  [page 43] Start an ftp client.
- `pwd(Pid) -> {ok, Dir} | {error, Reason}`
  [page 45] Get remote current working directory.
- `recv(Pid, RemoteFile [, LocalFile]) -> ok | {error, Reason}`
  [page 45] Transfer file from remote server.
- `recv_bin(Pid, RemoteFile) -> {ok, Bin} | {error, Reason}`
  [page 45] Transfer file from remote server as a binary.
- `recv_chunk_start(Pid, RemoteFile) -> ok | {error, Reason}`
  [page 46] Start chunk-reading of the remote file.
- `recv_chunk(Pid) -> ok | {ok, Bin} | {error, Reason}`
  [page 46] Receive a chunk of the remote file.
- `rename(Pid, Old, New) -> ok | {error, Reason}`
  [page 46] Rename a file at the remote server..
- `rmdir(Pid, Dir) -> ok | {error, Reason}`
  [page 46] Remove a remote directory.
- `send(Pid, LocalFile [, RemoteFile]) -> ok | {error, Reason}`
  [page 46] Transfer file to remote server.
- `send_bin(Pid, Bin, RemoteFile) -> ok | {error, Reason}`
  [page 47] Transfer a binary into a remote file.
- `send_chunk(Pid, Bin) -> ok | {error, Reason}`
  [page 47] Write a chunk to the remote file.
- `send_chunk_start(Pid, File) -> ok | {error, Reason}`
  [page 47] Start transfer of file chunks.

- send_chunk_end(Pid) -> ok | {error, Reason}
  [page 47] Stop transfer of chunks.
- type(Pid, Type) -> ok | {error, Reason}
  [page 47] Set transfer type to ascii or binary.
- user(Pid, User, Password) -> ok | {error, Reason}
  [page 47] User login.
- user(Pid, User, Password,Account) -> ok | {error, Reason}
  [page 48] User login.
- quote(Pid, Command) -> [FTPLine]
  [page 48] Sends an arbitary FTP command.

http

The following functions are exported:

- cancel_request(RequestId) -> ok
  [page 51] Cancels an asynchronous HTTP-request.
- request(Url) -> {ok, Result} | {error, Reason}
  [page 51] Sends a get HTTP-request
- request(Method, Request, HTTPOptions, Options) -> {ok, Result} | {ok, saved_to_file} | {error, Reason}
  [page 51] Sends a HTTP-request
- set_options(Options) -> ok
  [page 52] Sets options to be used for subsequent requests.
- verify_cookie(SetCookieHeaders, Url) -> ok
  [page 53] Saves the cookies defined in SetCookieHeaders in the client manager's cookie database.
- cookie_header(Url) -> header()
  [page 54] Returns the cookie header that would be sent when making a request to Url.

http_base_64

The following functions are exported:

- encode(PlainASCII) -> Base64
  [page 55] Encodes a plain ASCII string into base64.
- decode(Base64) -> PlainASCII
  [page 55] Decodes an base64 encoded string to plain ASCII.

httpd

The following functions are exported:

- `Module:do(ModData)-> {proceed, OldData} | {proceed, NewData} | {break, NewData} | done`
  [page 57] Called for each request to the Web server.
- `Module:load(Line, Context)-> eof | ok | {ok, NewContext} | {ok, NewContext, Directive} | {ok, NewContext, DirectiveList} | {error, Reason}`
  [page 58] Load a configuration directive.
- `Module:store({DirectiveKey, DirectiveValue}, DirectiveList)-> {ok, {DirectiveKey, NewDirectiveValue}} | {ok, [{ok, {DirectiveKey, NewDirectiveValue}} | {error, Reason}`
  [page 58] Alter the value of one or more configuration directive.
- `Module:remove(ConfigDB)-> ok | {error, Reason}`
  [page 58] Callback function that is called when the Web server is closed.
- `parse_query(QueryString) -> ServerRet`
  [page 59] Parse incoming data to `erl` and `eval` scripts.
- `start()`
  [page 59] Start a server as specified in the given config file.
- `start(Config) -> ServerRet`
  [page 59] Start a server as specified in the given config file.
- `start_link()`
  [page 59] Start a server as specified in the given config file.
- `start_link(Config) -> ServerRet`
  [page 59] Start a server as specified in the given config file.
- `restart()`
  [page 60] Restart a running server.
- `restart(Port) -> ok | {error,Reason}`
  [page 60] Restart a running server.
- `restart(ConfigFile) -> ok | {error,Reason}`
  [page 60] Restart a running server.
- `restart(Address,Port) -> ok | {error,Reason}`
  [page 60] Restart a running server.
- `stop()`
  [page 60] Stop a running server.
- `stop(Port) -> ServerRet`
  [page 60] Stop a running server.
- `stop(ConfigFile) -> ServerRet`
  [page 60] Stop a running server.
- `stop(Address,Port) -> ServerRet`
  [page 60] Stop a running server.
- `block() -> ok | {error,Reason}`
  [page 60] Block a running server.
- `block(Port) -> ok | {error,Reason}`
  [page 60] Block a running server.

- block(ConfigFile) -> ok | {error,Reason}
  [page 60] Block a running server.
- block(Address,Port) -> ok | {error,Reason}
  [page 60] Block a running server.
- block(Port,Mode) -> ok | {error,Reason}
  [page 60] Block a running server.
- block(ConfigFile,Mode) -> ok | {error,Reason}
  [page 60] Block a running server.
- block(Address,Port,Mode) -> ok | {error,Reason}
  [page 60] Block a running server.
- block(ConfigFile,Mode,Timeout) -> ok | {error,Reason}
  [page 60] Block a running server.
- block(Address,Port,Mode,Timeout) -> ok | {error,Reason}
  [page 60] Block a running server.
- unblock() -> ok | {error,Reason}
  [page 61] Unblock a blocked server.
- unblock(Port) -> ok | {error,Reason}
  [page 61] Unblock a blocked server.
- unblock(ConfigFile) -> ok | {error,Reason}
  [page 61] Unblock a blocked server.
- unblock(Address,Port) -> ok | {error,Reason}
  [page 61] Unblock a blocked server.

httpd_conf

The following functions are exported:

- check_enum(EnumString,ValidEnumStrings) -> Result
  [page 62] Check if string is a valid enumeration.
- clean(String) -> Stripped
  [page 62] Remove leading and/or trailing white spaces.
- custom_clean(String,Before,After) -> Stripped
  [page 62] Remove leading and/or trailing white spaces and custom characters.
- is_directory(FilePath) -> Result
  [page 62] Check if a file path is a directory.
- is_file(FilePath) -> Result
  [page 63] Check if a file path is a regular file.
- make_integer(String) -> Result
  [page 63] Return an integer representation of a string.

httpd_socket

The following functions are exported:

- deliver(SocketType, Socket, Data) -> Result
  [page 64] Send binary data over socket.
- peername(SocketType,Socket) -> {Port,IPAddress}
  [page 64] Return the port and IP-address of the remote socket.
- resolve() -> HostName
  [page 64] Return the official name of the current host.

httpd_util

The following functions are exported:

- `convert_request_date(DateString) -> ErlDate|bad_date`
  [page 65] Convert The the date to the Erlang date format.
- `create_etag(FileInfo) -> Etag`
  [page 65] Calculates the Etag for a file.
- `decode_base64(Base64String) -> ASCIIString`
  [page 65] Convert a base64 encoded string to a plain ascii string.
- `decode_hex(HexValue) -> DecValue`
  [page 65] Convert a hex value into its decimal equivalent.
- `day(NthDayOfWeek) -> DayOfWeek`
  [page 65] Convert the day of the week (integer [1-7]) to an abbreviated string.
- `encode_base64(ASCIIString) -> Base64String`
  [page 66] Convert an ASCII string to a Base64 encoded string.
- `flatlength(NestedList) -> Size`
  [page 66] Compute the size of a possibly nested list.
- `header(StatusCode,PersistentConn)`
  [page 66] Generate a HTTP 1.1 header.
- `header(StatusCode,Date)`
  [page 66] Generate a HTTP 1.1 header.
- `header(StatusCode,MimeType,Date)`
  [page 66] Generate a HTTP 1.1 header.
- `header(StatusCode,MimeType,PersistentConn,Date) -> HTTPHeader`
  [page 66] Generate a HTTP 1.1 header.
- `hexlist_to_integer(HexString) -> Number`
  [page 66] Convert a hexadecimal string to an integer.
- `integer_to_hexlist(Number) -> HexString`
  [page 66] Convert an integer to a hexadecimal string.
- `key1search(TupleList,Key)`
  [page 67] Search a list of key-value tuples for a tuple whose first element is a key.
- `key1search(TupleList,Key,Undefined) -> Result`
  [page 67] Search a list of key-value tuples for a tuple whose first element is a key.
- `lookup(ETSTable,Key) -> Result`
  [page 67] Extract the first value associated with a key in an ETS table.
- `lookup(ETSTable,Key,Undefined) -> Result`
  [page 67] Extract the first value associated with a key in an ETS table.
- `lookup_mime(ConfigDB,Suffix)`
  [page 67] Return the mime type associated with a specific file suffix.
- `lookup_mime(ConfigDB,Suffix,Undefined) -> MimeType`
  [page 67] Return the mime type associated with a specific file suffix.
- `lookup_mime_default(ConfigDB,Suffix)`
  [page 67] Return the mime type associated with a specific file suffix or the value of the DefaultType.
- `lookup_mime_default(ConfigDB,Suffix,Undefined) -> MimeType`
  [page 67] Return the mime type associated with a specific file suffix or the value of the DefaultType.

- message(StatusCode,PhraseArgs,ConfigDB) -> Message
  [page 68] Return an informative HTTP 1.1 status string in HTML.
- month(NthMonth) -> Month
  [page 68] Convert the month as an integer (1-12) to an abbreviated string.
- multi_lookup(ETSTable,Key) -> Result
  [page 68] Extract the values associated with a key in a ETS table.
- reason_phrase(StatusCode) -> Description
  [page 68] Return the description of an HTTP 1.1 status code.
- rfc1123_date() -> RFC1123Date
  [page 69] Return the current date in RFC 1123 format.
- rfc1123_date({{YYYY,MM,DD},{Hour,Min,Sec}}) -> RFC1123Date
  [page 69] Return the current date in RFC 1123 format.
- split(String,RegExp,N) -> SplitRes
  [page 69] Split a string in N chunks using a regular expression.
- split_script_path(RequestLine) -> Splitted
  [page 69] Split a RequestLine in a file reference to an executable and a QueryString or a PathInfo string.
- split_path(RequestLine) -> {Path,QueryStringOrPathInfo}
  [page 69] Split a RequestLine in a file reference and a QueryString or a PathInfo string.
- strip(String) -> Stripped
  [page 69] Returns String where the leading and trailing space and tabs has been removed.
- suffix(FileName) -> Suffix
  [page 70] Extract the file suffix from a given filename.
- to_lower(String) -> ConvertedString
  [page 70] Convert upper-case letters to lower-case.
- to_upper(String) -> ConvertedString
  [page 70] Convert lower-case letters to upper-case.

mod_alias

The following functions are exported:

- default_index(ConfigDB, Path) -> NewPath
  [page 71] Return a new path with the default resource or file appended.
- path(PathData, ConfigDB, RequestURI) -> Path
  [page 71] Return the actual file path to a URL.
- real_name(ConfigDB, RequestURI, Aliases) -> Ret
  [page 71] Expand a request uri using Alias config directives.
- real_script_name(ConfigDB,RequestURI,ScriptAliases) -> Ret
  [page 72] Expand a request uri using ScriptAlias config directives.

mod_auth

The following functions are exported:

- `add_user(UserName, Options) -> true| {error, Reason}`
  [page 73] Add a user to the user database.
- `add_user(UserName, Password, UserData, Port, Dir) -> true | {error, Reason}`
  [page 73] Add a user to the user database.
- `add_user(UserName, Password, UserData, Address, Port, Dir) -> true | {error, Reason}`
  [page 73] Add a user to the user database.
- `delete_user(UserName,Options) -> true | {error, Reason}`
  [page 73] Delete a user from the user database.
- `delete_user(UserName, Port, Dir) -> true | {error, Reason}`
  [page 73] Delete a user from the user database.
- `delete_user(UserName, Address, Port, Dir) -> true | {error, Reason}`
  [page 73] Delete a user from the user database.
- `get_user(UserName,Options) -> {ok, #httpd_user} |{error, Reason}`
  [page 74] Returns a user from the user database.
- `get_user(UserName, Port, Dir) -> {ok, #httpd_user} | {error, Reason}`
  [page 74] Returns a user from the user database.
- `get_user(UserName, Address, Port, Dir) -> {ok, #httpd_user} | {error, Reason}`
  [page 74] Returns a user from the user database.
- `list_users(Options) -> {ok, Users} | {error, Reason}`
  `<name>list_users(Port, Dir) -> {ok, Users} | {error, Reason}`
  [page 74] List users in the user database.
- `list_users(Address, Port, Dir) -> {ok, Users} | {error, Reason}`
  [page 74] List users in the user database.
- `add_group_member(GroupName, UserName, Options) -> true | {error, Reason}`
  [page 74] Add a user to a group.
- `add_group_member(GroupName, UserName, Port, Dir) -> true | {error, Reason}`
  [page 74] Add a user to a group.
- `add_group_member(GroupName, UserName, Address, Port, Dir) -> true | {error, Reason}`
  [page 74] Add a user to a group.
- `delete_group_member(GroupName, UserName, Options) -> true | {error, Reason}`
  [page 75] Remove a user from a group.
- `delete_group_member(GroupName, UserName, Port, Dir) -> true | {error, Reason}`
  [page 75] Remove a user from a group.
- `delete_group_member(GroupName, UserName, Address, Port, Dir) -> true | {error, Reason}`
  [page 75] Remove a user from a group.

- list_group_members(GroupName, Options) -> {ok, Users} | {error, Reason}
  [page 75] List the members of a group.
- list_group_members(GroupName, Port, Dir) -> {ok, Users} | {error, Reason}
  [page 75] List the members of a group.
- list_group_members(GroupName, Address, Port, Dir) -> {ok, Users} | {error, Reason}
  [page 75] List the members of a group.
- list_groups(Options) -> {ok, Groups} | {error, Reason}
  [page 76] List all the groups.
- list_groups(Port, Dir) -> {ok, Groups} | {error, Reason}
  [page 76] List all the groups.
- list_groups(Address, Port, Dir) -> {ok, Groups} | {error, Reason}
  [page 76] List all the groups.
- delete_group(GroupName, Options) -> true | {error,Reason}
  <name>delete_group(GroupName, Port, Dir) -> true | {error, Reason}
  [page 76] Deletes a group
- delete_group(GroupName, Address, Port, Dir) -> true | {error, Reason}
  [page 76] Deletes a group
- update_password(Port, Dir, OldPassword, NewPassword, NewPassword) -> ok | {error, Reason}
  [page 76] Change the AuthAcessPassword
- update_password(Address,Port, Dir, OldPassword, NewPassword, NewPassword) -> ok | {error, Reason}
  [page 76] Change the AuthAcessPassword

mod_esi

The following functions are exported:

- deliver(SessionID, Data) -> ok | {error, Reason}
  [page 78] Sends Data back to client.
- Module:Function(SessionID, Env, Input)-> _
  [page 78] Creates a dynamic web page and returns it chunk by chunk to the server process by calling mod_esi:deliver/2.
- Module:Function(Env, Input)-> Response
  [page 79] Creates a dynamic web page and return it as a list. This functions is deprecated and only keept for backwards compability.

mod_security

The following functions are exported:

- list_auth_users(Port) -> Users | []
  [page 80] List users that have authenticated within the SecurityAuthTimeout time for a given address (if specified), port number and directory (if specified).

- `list_auth_users(Address, Port) -> Users | []`
  [page 80] List users that have authenticated within the SecurityAuthTimeout time for a given address (if specified), port number and directory (if specified).
- `list_auth_users(Port, Dir) -> Users | []`
  [page 80] List users that have authenticated within the SecurityAuthTimeout time for a given address (if specified), port number and directory (if specified).
- `list_auth_users(Address, Port, Dir) -> Users | []`
  [page 80] List users that have authenticated within the SecurityAuthTimeout time for a given address (if specified), port number and directory (if specified).
- `list_blocked_users(Port) -> Users | []`
  [page 80] List users that are currently blocked from access to a specified port number, for a given address (if specified).
- `list_blocked_users(Address, Port) -> Users | []`
  [page 80] List users that are currently blocked from access to a specified port number, for a given address (if specified).
- `list_blocked_users(Port, Dir) -> Users | []`
  [page 80] List users that are currently blocked from access to a specified port number, for a given address (if specified).
- `list_blocked_users(Address, Port, Dir) -> Users | []`
  [page 80] List users that are currently blocked from access to a specified port number, for a given address (if specified).
- `block_user(User, Port, Dir, Seconds) -> true | {error, Reason}`
  [page 80] Block user from access to a directory for a certain amount of time.
- `block_user(User, Address, Port, Dir, Seconds) -> true | {error, Reason}`
  [page 80] Block user from access to a directory for a certain amount of time.
- `unblock_user(User, Port) -> true | {error, Reason}`
  [page 81] Remove a blocked user from the block list
- `unblock_user(User, Address, Port) -> true | {error, Reason}`
  [page 81] Remove a blocked user from the block list
- `unblock_user(User, Port, Dir) -> true | {error, Reason}`
  [page 81] Remove a blocked user from the block list
- `unblock_user(User, Address, Port, Dir) -> true | {error, Reason}`
  [page 81] Remove a blocked user from the block list
- `event(What, Port, Dir, Data) -> ignored`
  [page 81] This function is called whenever an event occurs in mod_security
- `event(What, Address, Port, Dir, Data) -> ignored`
  [page 81] This function is called whenever an event occurs in mod_security

tftp

The following functions are exported:

- `start(Options) -> {ok, Pid} | {error, Reason}`
  [page 85] Start a daemon process
- `read_file(RemoteFilename, LocalFilename, Options) -> {ok, LastCallbackState} | {error, Reason}`
  [page 85] Read a (virtual) file from a TFTP server

- `write_file(RemoteFilename, LocalFilename, Options) -> {ok, LastCallbackState} | {error, Reason}`
  [page 85] Write a (virtual) file to a TFTP server
- `info(Pid) -> undefined | Options`
  [page 86] Return information about a daemon, server or client process
- `start() -> ok | {error, Reason}`
  [page 86] Start the Inets application
- `prepare(Peer, Access, Filename, Mode, SuggestedOptions, InitialState) -> {ok, AcceptedOptions, NewState} | {error, {Code, Text}}`
  [page 87] Prepare to open a file on the client side
- `open(Peer, Access, Filename, Mode, SuggestedOptions, State) -> {ok, AcceptedOptions, NewState} | {error, {Code, Text}}`
  [page 87] Open a file for read or write access
- `read(State) -> {more, Bin, NewState} | {last, Bin, FileSize} | {error, {Code, Text}}`
  [page 88] Read a chunk from the file
- `write(Bin, State) -> {more, NewState} | {last, FileSize} | {error, {Code, Text}}`
  [page 88] Write a chunk to the file
- `abort(Code, Text, State) -> ok`
  [page 88] Abort the file transfer

ftp

Erlang Module

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

For a simple example of an ftp session see Inets User's Guide. [page 2]

In addition to the ordinary functions for receiving and sending files (see recv/2, recv/3, send/2 and send/3) there are functions for receiving remote files as binaries (see recv_bin/2) and for sending binaries to to be stored as remote files (see send_bin/3).

There is also a set of functions for sending and receiving contiguous parts of a file to be stored in a remote file (for send see send_chunk_start/2, send_chunk/2 and send_chunk_end/1 and for receive see recv_chunk_start/2 and recv_chunk/).

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

COMMON DATA TYPES

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

```
pid() - identifier of an ftp connection.

string() = list of ASCII characters

shortage_reason() = etnospc | epnospc

restriction_reason() = epath | efnamena | elogin | enotbinary
    - note not all restrictions may always relevant to all functions

common_reason() = econn | eclosed | term() - some kind of
                  explanation of what went wrong
```

Exports

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

Types:

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

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

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

Types:

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

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

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

Types:

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

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

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

Types:

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

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

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

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

Types:

- Pid = pid()
- File = string()
- Reason = restriction_reason() | common_reason()

Start the transfer of chunks for appending to the file `File` at the remote server. If the file does not exists it will be created.

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

Types:

- Pid = pid()
- Reason = echunk | restriction_reason() | shortage_reason()

Stops transfer of chunks for appending to the remote server. The file at the remote server, specified in the call to `append_chunk_start/2` is closed by the server.

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

Types:

- Pid = pid()
- Dir = string()
- Reason = restriction_reason() | common_reason()

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 = restriction_reason() | common_reason()

Deletes the file `File` at the remote server.

`formaterror(Tag) -> string()`

Types:

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

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

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

Types:

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

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 = restriction_reason() | common_reason()

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 = restriction_reason() | common_reason()

Creates the directory `Dir` at the remote server.

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

Types:

- Pid = pid()
- Dir = string()
- Listing = string()
- Reason = restriction_reason() | common_reason()

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

Types:

- Host = string() | ip_address()
- ip_address() = {byte(), byte(), byte(), byte()} {byte(), byte(), byte(), byte(), byte(), byte(), byte(), byte()}
- byte() = 0 | 1 | ... | 255
- Port = integer()
- Flags = [Flag]
- Flag = verbose | debug | ip_v6_disabled
- Pid = pid()

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

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

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

Default value for `Mode` is passive.

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

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

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

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

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

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

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

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

The callback function should be defined as

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

CBModule = CBFunction = atom()

UserProgressTerm = term()

File = string()

Size = {transfer_size, integer()} | {file_size, integer()} | {file_size, unknown}
```

Alas for remote files it is not possible for ftp to determine the file size in a platform independent way. In this case the size will be `unknown` and it is left to the application to find out the size.

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

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

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

The timeout value is default set to 60000 milliseconds.

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

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

Types:

- Pid = pid()
- Reason = restriction_reason() | common_reason()

Returns the current working directory at the remote server.

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

Types:

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

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

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

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

Types:

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

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

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

Types:

- Pid = pid()
- RemoteFile = string()
- Reason = restriction_reason() | common_reason()

Start transfer of the file RemoteFile from the remote server.

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

Types:

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

Receive a chunk of the remote file (RemoteFile of recv_chunk_start). The return values has the following meaning:

- ok the transfer is complete.
- {ok, Bin} just another chunk of the file.
- {error, Reason} transfer failed.

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

Types:

- Pid = pid()
- CurrFile = NewFile = string()
- Reason = restriction_reason() | common_reason()

Renames Old to New at the remote server.

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

Types:

- Pid = pid()
- Dir = string()
- Reason = restriction_reason() | common_reason()

Removes directory Dir at the remote server.

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

Types:

- Pid = pid()
- LocalFile = RemoteFile = string()
- Reason = restriction_reason() | common_reason() | shortage_reason()

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 = restriction_reason() | common_reason() | shortage_reason()

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 = echunk | restriction_reason() | common_reason()

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

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

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

Types:

- Pid = pid()
- File = string()
- Reason = restriction_reason() | common_reason()

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

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

Types:

- Pid = pid()
- Reason = restriction_reason() | common_reason() | shortage_reason()

Stops transfer of chunks to the remote server. The file at the remote server, specified in the call to `send_chunk_start/2` is closed by the server.

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

Types:

- Pid = pid()
- Type = ascii | binary
- Reason = etype | restriction_reason() | common_reason()

Sets the file transfer type to `ascii` or `binary`. When an ftp session is opened, the default transfer type of the server is used, most often `ascii`, which is the default according to RFC 959.

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

Types:

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

Performs login of `User` with `Password`.

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

Types:

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

Performs login of `User` with `Passwordto` the acccount specified by `Account` .

`quote(Pid, Command) -> [FTPLine]`

Types:

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

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

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

ERRORS

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

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

`eclosed` The session has been closed.

`econn` Connection to remote server prematurely closed.

`ehost` Host not found, FTP server not found, or connection rejected by FTP server.

`elogin` User not logged in.

`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.

`etnospc` Insufficient storage space in system [452].

`epnospc` Exceeded storage allocation (for current directory or dataset) [552].

`efnamena` File name not allowed [553].

SEE ALSO

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

http

Erlang Module

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

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

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

COMMON DATA TYPES

Type definitions that are used more than once in this module:

```
boolean() = true | false

string() = list of ASCII characters

request_id() = ref()
```

HTTP DATA TYPES

Type definitions that are related to HTTP:

For more information about HTTP see rfc 2616

```
method() = head | get | put | post | trace | options | delete

request() - {url(), headers()} |
     {url(), headers(), content_type(), body()}

url() = string() - Syntax according to the URI definition in rfc 2396, ex: "http://www.er

status_line() =
     {http_version(), status_code(), reason_phrase()}

http_version() = string() ex: "HTTP/1.1"

status_code() = integer()
```

```
reason_phrase() = string()

content_type() = string()

headers() = [{field(), value()}]

field() = string()

value() = string()

body() = string() | binary()

filename = string()
```

SSL DATA TYPES

Some type definitions relevant when using https, for details [ssl(3)]:

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

Exports

`cancel_request(RequestId) -> ok`

Types:

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

Cancels an asynchronous HTTP-request.

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

Types:

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

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

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

Types:

- Method = method()
- Request - request()
- HTTPOptions - [HttpOption]
- HTTPOption - {timeout, integer()} | {ssl, ssl_options()} | {autoredirect, boolean()} | {proxy_auth, {userstring(), passwordstring()}}

- autoredirect
  This option is true by default i.e. the client will automatically retrive the information from the new URI and return that as the result instead of a 30X-result code. Note that for some 30X-result codes automatic redirect is not allowed in these cases the 30X-result will always be returned.
- proxy_auth
  A proxy-authorization header using the provided user name and password will be added to the request.
- Options - [option()]
- Option - {sync, boolean()} | {stream, StreamTo} | {body_format, body_format()} | {full_result, boolean()} | {headers_as_is, boolean()}
  The request function will be synchronous and return a full http response by default.
- StreamTo = self | filename()
  Streams the body of a 200 response to the calling process or to a file. When streaming to the calling process the the following stream messages will be sent to that process: {http, {RequestId, stream_start, Headers}, {http, {RequestId, stream, BinBodyPart}, {http, {RequestId, stream_end, Headers}. Note that it is possible that chunked encoding will add headers so that there are more headers in the stream_end message than in the stream_start. When streaming to a file and the request is asynchronous the message {http, {RequestId, saved_to_file}} will be sent.
- body_format() = string() | binary()
  The body_format option is only valid for the synchronous request and the default is string. When making an asynchronous request the body will always be received as a binary.
- headers_as_is
  The headers_as_is option is by default false, if set to true the headers provided by the user will be regarded as case sensative. Note that the http standard requires them to be case insensative. This feature should only be used if there is no other way to communicate with the server or for testing purpose. Also note that when this option is used no headers will be automaticly added, all necessary headers has to be provided by the user.
- Result = {status_line(), headers(), body()} | {status_code(), body()} | request_id()
- Reason = term()

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

`set_options(Options) -> ok`

Types:

- Options = [Option]
- Option = {proxy, {Proxy, NoProxy}} | {max_sessions, MaxSessions} | {max_pipeline_length, MaxPipeline} | {pipeline_timeout, PipelineTimeout} | {cookies | CookieMode} | {ipv6, Ipv6Mode} | {verbose, VerboseMode}
- Proxy = {Hostname, Port}
- Hostname = string()
  ex: ”localhost” or ”foo.bar.se”

- Port = integer()
  ex: 8080
- NoProxy = [NoProxyDesc]
- NoProxyDesc = DomainDesc | HostName | IPDesc
- DomainDesc = "*.Domain"
  ex: "*.ericsson.se"
- IpDesc = string()
  ex: "134.138" or "[FEDC:BA98" (all IP-adresses starting with 134.138 or FEDC:BA98), "66.35.250.150" or "[2010:836B:4179::836B:4179]" (a complete IP-address).
- MaxSessions = integer()
  Maximum number of persistent connections to a host.Default is 2.
- MaxPipeline = integer()
  Maximum number of outstanding requests on the same connection to a host. Default is 2.
- PipelineTimeout = integer()
  If a persistent connection is idle longer than the pipeline_timeout the client will close the connection. Default is 0. The server may also have a such a time out but you should not count on it!
- CookieMode = enabled | disabled | verify
  Default is disabled. If Cookies are enabled all valid cookies will automatically be saved in the client manager's cookie database. If the option verify is used the function http:verify_cookie/2 has to be called for the cookie to be saved.
- ipv6Mode = enabled | disabled
  By default enabled. This should normally be what you want. When it is enabled you can use both ipv4 and ipv6. The option is here to provide a workaround for buggy ipv6 stacks to ensure that ipv4 will always work.
- VerboseMode = false | verbose |debug | trace
  By default false. This option unless it is set to false switches on different levels of erlang trace on the client. It is a debug feature.

Sets options to be used for subsequent requests. Later implementations may support user profiles, but currently these are global settings for all clients running on the same erlang node.

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

`verify_cookie(SetCookieHeaders, Url) -> ok`

Types:

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

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

`cookie_header(Url) -> header()`

Types:

- Url = url()

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

SEE ALSO

RFC 2616, [ssl(3)]

http_base_64

Erlang Module

Implements base 64 encode and decode, see RFC2045.

COMMON DATA TYPES

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

```
string() = list of ASCII characters
```

Exports

`encode(PlainASCII) -> Base64`

Types:

- PlainASCII = string()
- Base64 = string()

Encodes a plain ASCII string into base64.

`decode(Base64) -> PlainASCII`

Types:

- PlainASCII = string()
- Base64 = string()

Decodes an base64 encoded string to plain ASCII.

httpd

Erlang Module

Mainly specifies the Erlang Webserver callback API but also provide some test and admin. functionality.

DATA TYPES

```
ModData = #mod{}

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

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

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

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

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

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

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

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

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

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

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

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

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

`connection true | false` If set to true the connection to the client is a persistent connections and will not be closed when the request is served.

Erlang Webserver API Callback Functions

Exports

`Module:do(ModData)-> {proceed, OldData} | {proceed, NewData} | {break, NewData} | done`

Types:

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

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

The field `data` in ModData is a list. This list will be the list returned from the from the last call to `do/1`.

`Body` is the body of the http-response that will be sent back to the client an appropriate header will be appended to the message. `StatusCode` will be the status code of the response see RFC2616 for the appropriate values.

`Head` is a key value list of HTTP header fields. the server will construct a HTTP header from this data. See RFC 2616 for the appropriate value for each header field. If the

client is a HTTP/1.0 client then the server will filter the list so that only HTTP/1.0 header fields will be sent back to the client.

If Body2 is returned and equal to {Fun,Arg} The Web server will try apply/2. on Fun with Arg as argument and excpect that the fun either returns a list (Body) that is a HTTP-repsonse or the atom sent if the HTTP-response is sent back to the client. If close is returned from the fun something has gone wrong and the server will signal this to the client by closing the connection.

```
Module:load(Line, Context)-> eof | ok | {ok, NewContext} | {ok, NewContext,
            Directive} | {ok, NewContext, DirectiveList} | {error, Reason}
```

Types:

- Line = string()
- Context = NewContext = DirectiveList = [Directive]
- Directive = {DirectiveKey , DirectiveValue}
- DirectiveKey = DirectiveValue = term()
- Reason = term()

load/2 takes a row Line from the configuration file and tries to convert it to a key value tuple. If a directive is dependent on other directives, the directive may create a context. If the directive is not dependent on other directives return {ok, [], Directive}, otherwise return a new context, that is {ok, NewContext} or {ok, Context Directive}. If {error, Reason} is returned the configuration directive is assumed to be invalid.

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

Types:

- DirectiveList = [{DirectiveKey, DirectiveValue}]
- DirectiveKey = DirecitveValue = term()
- Context = NewContext = DirectiveList = [Directive]
- Directive = {Key , Value}
- Reason = term()

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

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

Types:

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

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

Erlang Webserver API Help Functions

Exports

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 78]) as defined in the standard URL format, that is '+' becomes 'space' and decoding of hexadecimal characters (%xx).

Erlang Webserver Test and Admin Functions

Exports

start()
start(Config) -> ServerRet
start_link()
start_link(Config) -> ServerRet

Types:

- Config = string() | HttpdOptions
- ServerRet = {ok,Pid} | ignore | {error,EReason} | {stop,SReason}
- Pid = pid()
- EReason = {already_started, Pid} | term()
- SReason = string()
- HttpdOptions = {file,string()} , {debug,Debug}? , {accept_timeout,integer()}?
- Debug = disable | [DebugOptions]
- DebugOptions = {all_functions,Modules} | {exported_functions,Modules} | {disable,Modules}
- Modules = [atom()]

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

start/0 and 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 or start_link/0, copy the example server root[1] to a specific installation directory[2] 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").

[1] In Windows: %INETS_ROOT%\examples\server_root\. In UNIX: $INETS_ROOT/examples/server_root/.
[2] In Windows: X:\var\tmp\. In UNIX: /var/tmp/.

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

Types:

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

`restart` restarts the server and reloads its config file.

The follwing directives cannot be changed: BindAddress, Port and SocketType. If these should be changed, then a new server should be started instead.

> **Note:**
> Before the `restart` function can be called the server must be blocked [page 61]. After `restart` has been called, the server must be unblocked [page 61].

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

Types:

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

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

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

Types:

- Port = integer()

- Address = {A,B,C,D} | string() | undefined
- ConfigFile = string()
- Mode = disturbing | non_disturbing
- Timeout = integer()
- Reason = term()

This function is used to block a server. The blocking can be done in two ways, disturbing or non-disturbing.

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

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

Default mode is disturbing.

Default port is 8888

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

Types:

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

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

httpd_conf

Erlang Module

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

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 | FileInfo
- FileInfo = File info record

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

is_file(FilePath) -> Result

Types:

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

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

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 56]

httpd_socket

Erlang Module

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

Exports

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

Types:

- SocketType = socket_type()
- Socket = socket()
- Data = io_list() | binary()
- Result = socket_closed | void()

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

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

Types:

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

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

`resolve() -> HostName`

Types:

- HostName = string()

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

SEE ALSO

httpd(3) [page 56]

httpd_util

Erlang Module

This module provides the Erlang Web Server API module programmer with miscellaneous utility functions.

Exports

`convert_request_date(DateString) -> ErlDate|bad_date`

Types:

- DateString = string()
- ErlDate = {{Year,Month,Date},{Hour,Min,Sec}}
- Year = Month = Date = Hour = Min = Sec = integer()

`convert_request_date/1` converts `DateString` to the Erlang date format. DateString must be in one of the three date formats that is defined in the RFC 2616.

`create_etag(FileInfo) -> Etag`

Types:

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

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

`decode_base64(Base64String) -> ASCIIString`

Types:

- Base64String = ASCIIString = string()

Deprecated use http_base_64:decode/1 [page 55]

`decode_hex(HexValue) -> DecValue`

Types:

- HexValue = DecValue = string()

Converts the hexadecimal value `HexValue` into it's 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(ASCIIString) -> Base64String

Types:

- ASCIIString = string()
- Base64String = string()

Deprecated use http_base_64:decode/1 [page 55]

flatlength(NestedList) -> Size

Types:

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

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

header(StatusCode,PersistentConn)

header(StatusCode,Date)

header(StatusCode,MimeType,Date)

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

Types:

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

header returns a HTTP 1.1 header string. The StatusCode is one of the status codes defined in RFC 2616 and the Date string is RFC 1123 compliant. (See rfc1123_date/0 [page 69]).

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

hexlist_to_integer(HexString) -> Number

Types:

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

hexlist_to_integer Convert the Hexadecimal value of HexString to an integer.

integer_to_hexlist(Number) -> HexString

Types:

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

integer_to_hexlist/1 Returns a string that represents the Number in a Hexadecimal form.

```
key1search(TupleList,Key)
key1search(TupleList,Key,Undefined) -> Result
```

Types:

- TupleList = [tuple()]
- Key = term()
- Result = term() | undefined | Undefined
- Undefined = term()

key1search searches the TupleList for a tuple whose first element is Key. key1search/2 returns undefined and key1search/3 returns Undefined if no tuple is found.

```
lookup(ETSTable,Key) -> Result
lookup(ETSTable,Key,Undefined) -> Result
```

Types:

- ETSTable = ets_table()
- Key = term()
- Result = term() | undefined | Undefined
- Undefined = term()

lookup extracts {Key,Value} tuples from ETSTable and returns the Value associated with Key. If ETSTable is of type bag only the first Value associated with Key is returned. lookup/2 returns undefined and lookup/3 returns Undefined if no Value is found.

```
lookup_mime(ConfigDB,Suffix)
lookup_mime(ConfigDB,Suffix,Undefined) -> MimeType
```

Types:

- ConfigDB = ets_table()
- Suffix = string()
- MimeType = string() | undefined | Undefined
- Undefined = term()

lookup_mime returns the mime type associated with a specific file suffix as specified in the mime.types file (located in the config directory[3]).

```
lookup_mime_default(ConfigDB,Suffix)
lookup_mime_default(ConfigDB,Suffix,Undefined) -> MimeType
```

Types:

- ConfigDB = ets_table()
- Suffix = string()
- MimeType = string() | undefined | Undefined
- Undefined = term()

[3] In Windows: %SERVER_ROOT%\conf\mime.types. In UNIX: $SERVER_ROOT/conf/mime.types.

lookup_mime_default returns the mime type associated with a specific file suffix as specified in the mime.types file (located in the config directory[4]). If no appropriate association can be found the value of DefaultType is returned.

message(StatusCode,PhraseArgs,ConfigDB) -> Message

Types:

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

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

301 string(): A URL pointing at the new document position.

400 | 401 | 500 none (No PhraseArgs)

403 | 404 string(): A Request-URI as described in RFC 2616.

501 {Method,RequestURI,HTTPVersion}: The HTTP Method, Request-URI and HTTP-Version as defined in RFC 2616.

504 string(): A string describing why the service was unavailable.

month(NthMonth) -> Month

Types:

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

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

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

multi_lookup(ETSTable,Key) -> Result

Types:

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

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

reason_phrase(StatusCode) -> Description

Types:

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

[4] In Windows: %SERVER_ROOT%\conf\mime.types. In UNIX: $SERVER_ROOT/conf/mime.types.

reason_phrase returns the Description of an HTTP 1.1 StatusCode, for example 200 is “OK” and 201 is “Created”. Read RFC 2616 for further information.

```
rfc1123_date() -> RFC1123Date
rfc1123_date({{YYYY,MM,DD},{Hour,Min,Sec}}) -> RFC1123Date
```

Types:

- YYYY = MM = DD = Hour = Min =Sec = integer()
- RFC1123Date = string()

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

```
split(String,RegExp,N) -> SplitRes
```

Types:

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

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

```
split_script_path(RequestLine) -> Splitted
```

Types:

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

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

```
split_path(RequestLine) -> {Path,QueryStringOrPathInfo}
```

Types:

- RequestLine = Path = QueryStringOrPathInfo = string()

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

```
strip(String) -> Stripped
```

Types:

- String = Stripped = string()

`strip/1` removes any leading or trailing linear white space from the string. Linear white space should be read as horisontal tab or space.

`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 56]

mod_alias

Erlang Module

Erlang Webserver Server internal API for handling of things such as interaction data exported by the mod_alias module.

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. 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 Inets Users Guide. [page 4].

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

Types:

- PathData = 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; `Path` is returned. If no interaction data has been exported, ServerRoot is used to generate a file `Path`. `config_db()` and `interaction_data()` are as defined in Inets Users Guide [page 4].

`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 69]. Path is generated from ShortPath, that is the result from default_index/2 [page 71] with ShortPath as an argument. config_db() is the server config file in ETS table format as described in Inets User Guide. [page 4].

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 69]. config_db() is the server config file in ETS table format as described in Inets Users Guide. [page 4].

mod_auth

Erlang Module

This module provides for basic user authentication using textual files, dets databases as well as mnesia databases.

Exports

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

Types:

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

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

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

Types:

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

- AuthPassword = string()
- Reason = term()

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

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

Types:

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

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

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

Types:

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

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

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

Types:

- GroupName = string()

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

`add_group_member/3`, `add_group_member/4` and `add_group_member/5` adds a user to a group. If the group does not exist, it is created and the user is added to the group. Upon successful operation, this function returns `true`. When `add_group_members/3` is called the Port and Dir options are mandatory.

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

Types:

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

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

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

Types:

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

- AuthPassword = string()
- Reason = term()

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

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

Types:

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

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

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

Types:

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

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

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

Types:

- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- Dir = string()
- GroupName = string()
- OldPassword = string()
- NewPassword = string()
- Reason = term()

`update_password/5` and `update_password/6` Updates the AuthAccessPassword for the specified directory. If NewPassword is equal to “NoPassword” no password is requires to change authorisation data. If NewPassword is equal to “DummyPassword” no changes can be done without changing the password first.

SEE ALSO

httpd(3) [page 56], mod_alias(3) [page 71],

mod_esi

Erlang Module

This module defines the API - Erlang Server Interface (ESI). Which is a more efficient way of writing erlang scripts for your Inets webserver than writing them as common CGI scripts.

Exports

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

Types:

- SessionID = term()
- Data = string() | io_list()
- Reason = term()

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

Sends data from a Erl Scheme script back to the client. Note that if any HTTP-header fields should be added by the script they must be in the first call to deliver/2 and the data in the call must be a string. Do not assume anything about the data type of SessionID, the SessionID must be the value given as input to the esi call back function that you implemented.

ESI Callback Functions

Exports

`Module:Function(SessionID, Env, Input)-> _`

Types:

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

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

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

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

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

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

`Module:Function(Env, Input)-> Response`

Types:

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

This callback format consumes quite much memory since the whole response must be generated before it is sent to the user.This functions is deprecated and only keept for backwards compability. For new development Module:Function/3 should be used.

mod_security

Erlang Module

Security Audit and Trailing Functionality

Exports

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

Types:

- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- Dir = string()
- Users = list() = [string()]

`list_auth_users/1`, `list_auth_users/2` and `list_auth_users/3` returns a list of users that are currently authenticated. Authentications are stored for SecurityAuthTimeout seconds, and are then discarded.

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

Types:

- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- Dir = string()
- Users = list() = [string()]

`list_blocked_users/1`, `list_blocked_users/2` and `list_blocked_users/3` returns a list of users that are currently blocked from access.

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

Types:

- User = string()
- Port = integer()
- Address = {A,B,C,D} | string() | undefined

- Dir = string()
- Seconds = integer() | infinity
- Reason = no_such_directory

`block_user/4` and `block_user/5` blocks the user `User` from the directory `Dir` for a specified amount of time.

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

Types:

- User = string()
- Port = integer()
- Address = {A,B,C,D} | string() | undefined
- Dir = string()
- Reason = term()

`unblock_user/2`, `unblock_user/3` and `unblock_user/4` removes the user `User` from the list of blocked users for the Port (and Dir) specified.

The SecurityCallbackModule

The SecurityCallbackModule is a user written module that can receive events from the mod_security Erlang Webserver API module. This module only exports one function, event/4 [page 81], which is described below.

Exports

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

Types:

- What = atom()
- Port = integer()
- Address = {A,B,C,D} | string() <v>Dir = string()
- What = [Info]
- Info = {Name, Value}

`event/4` or `event/4` is called whenever an event occurs in the mod_security Erlang Webserver API module (`event/4` is called if Address is undefined and `event/5` otherwise). 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.

tftp

Erlang Module

This is a complete implementation of the following IETF standards:

- RFC 1350, The TFTP Protocol (revision 2).
- RFC 2347, TFTP Option Extension.
- RFC 2348, TFTP Blocksize Option.
- RFC 2349, TFTP Timeout Interval and Transfer Size Options.

The only feature that not is implemented in this release is the “netascii” transfer mode.

The start/1 [page 85] function starts a daemon process which listens for UDP packets on a port. When it receives a request for read or write it spawns a temporary server process which handles the actual transfer of the file.

On the client side the read_file/3 [page 85] and write_file/3 [page 85] functions spawns a temporary client process which establishes contact with a TFTP daemon and performs the actual transfer of the file.

`tftp` uses a callback module to handle the actual file transfer. Two such callback modules are provided, `tftp_binary` and `tftp_file`. See read_file/3 [page 85] and write_file/3 [page 85] for more information about these. The user can also implement own callback modules, see CALLBACK FUNCTIONS [page 86] below. A callback module provided by the user is registered using the `callback` option, see DATA TYPES [page 83] below.

DATA TYPES

```
option() -- see below
```

Most of the options are common for both the client and the server side, but some of them differs a little. Here are the available options:

{`debug, Level`} `Level = none | brief | normal | verbose | all`

Controls the level of debug printouts. The default is `none`.

{`host, Host`} `Host = hostname()` see [inet(3)]

The name or IP address of the host where the TFTP daemon resides. This option is only used by the client.

{port, Port} Port = int()

The TFTP port where the daemon listens. It defaults to the standardized number 69. On the server side it may sometimes make sense to set it to 0, which means that the daemon just will pick a free port (which one is returned by the info/1 function).

If a socket has somehow already has been connected, the {udp, [{fd, integer()}]} option can be used to pass the open file descriptor to gen_udp. This can be automated a bit by using a command line argument stating the prebound file descriptor number. For example, if the Port is 69 and the file descriptor 22 has been opened by setuid_socket_wrap. Then the command line argument "-tftpd_69 22" will trigger the prebound file descriptor 22 to be used instead of opening port 69. The UDP option {udp, [{fd, 22}]} autmatically be added. See init:get_argument/ about command line arguments and gen_udp:open/2 about UDP options.

{port_policy, Policy} Policy = random | Port | {range, MinPort, MaxPort}
Port = MinPort = MaxPort = int()

Policy for the selection of the temporary port which is used by the server/client during the file transfer. It defaults to random which is the standardized policy. With this policy a randomized free port used. A single port or a range of ports can be useful if the protocol should pass through a firewall.

{udp, Options} Options = [Opt] see [gen_udp:open/2]

{use_tsize, Bool} Bool = bool()

Flag for automated usage of the tsize option. With this set to true, the write_file/3 client will determine the filesize and send it to the server as the standardized tsize option. A read_file/3 client will just acquire filesize from the server by sending a zero tsize.

{max_tsize, MaxTsize} MaxTsize = int() | infinity

Threshold for the maximal filesize in bytes. The transfer will be aborted if the limit is exceeded. It defaults to infinity.

{max_conn, MaxConn} MaxConn = int() | infinity

Threshold for the maximal number of active connections. The daemon will reject the setup of new connections if the limit is exceeded. It defaults to infinity.

{TftpKey, TftpVal} TftpKey = string()
TftpVal = string()

The name and value of a TFTP option.

{reject, Feature} Feature = Mode | TftpKey
Mode = read | write
TftpKey = string()

Control which features that should be rejected. This is mostly useful for the server as it may restrict usage of certain TFTP options or read/write access.

{callback, {RegExp, Module, State}} RegExp = string()
Module = atom()
State = term()

Registration of a callback module. When a file is to be transferred, its local filename will be matched to the regular expressions of the registered callbacks. The first matching callback will be used the during the transfer. See read_file/3 [page 85] and write_file/3 [page 85].

The callback module must implement the tftp behavior, CALLBACK FUNCTIONS [page 86].

Exports

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

Types:

- Options = [option()]
- Pid = pid()
- Reason = term()

Starts a daemon process which listens for udp packets on a port. When it receives a request for read or write it spawns a temporary server process which handles the actual transfer of the (virtual) file.

`read_file(RemoteFilename, LocalFilename, Options) -> {ok, LastCallbackState} | {error, Reason}`

Types:

- RemoteFilename = string()
- LocalFilename = binary | string()
- Options = [option()]
- LastCallbackState = term()
- Reason = term()

Reads a (virtual) file `RemoteFilename` from a TFTP server.

If `LocalFilename` is the atom `binary`, `tftp_binary` is used as callback module. It concatenates all transferred blocks and returns them as one single binary in `LastCallbackState`.

If `LocalFilename` is a string and there are no registered callback modules, `tftp_file` is used as callback module. It writes each transferred block to the file named `LocalFilename` and returns the number of transferred bytes in `LastCallbackState`.

If `LocalFilename` is a string and there are registered callback modules, `LocalFilename` is tested against the regexps of these and the callback module corresponding to the first match is used, or an error tuple is returned if no matching regexp is found.

`write_file(RemoteFilename, LocalFilename, Options) -> {ok, LastCallbackState} | {error, Reason}`

Types:

- RemoteFilename = string()
- LocalFilename = binary() | string()
- Options = [option()]
- LastCallbackState = term()
- Reason = term()

Writes a (virtual) file RemoteFilename to a TFTP server.

If LocalFilename is a binary, tftp_binary is used as callback module. The binary is transferred block by block and the number of transferred bytes is returned in LastCallbackState.

If LocalFilename is a string and there are no registered callback modules, tftp_file is used as callback module. It reads the file named LocalFilename block by block and returns the number of transferred bytes in LastCallbackState.

If LocalFilename is a string and there are registered callback modules, LocalFilename is tested against the regexps of these and the callback module corresponding to the first match is used, or an error tuple is returned if no matching regexp is found.

```
info(Pid) -> undefined | Options
```

Types:

- Options = [option()]

Returns info about a TFTP daemon, server or client process.

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

Types:

- Reason = term()

Starts the Inets application.

CALLBACK FUNCTIONS

A tftp callback module should be implemented as a tftp behavior and export the functions listed below.

On the server side the callback interaction starts with a call to open/5 with the registered initial callback state. open/5 is expected to open the (virtual) file. Then either the read/1 or write/2 functions are invoked repeatedly, once per transfererred block. At each function call the state returned from the previous call is obtained. When the last block has been encountered the read/1 or write/2 functions is expected to close the (virtual) file and return its last state. The abort/3 function is only used in error situations. prepare/5 is not used on the server side.

On the client side the callback interaction is the same, but it starts and ends a bit differently. It starts with a call to prepare/5 with the same arguments as open/5 takes. prepare/5 is expected to validate the TFTP options, suggested by the user and return the subset of them that it accepts. Then the options is sent to the server which will perform the same TFTP option negotiation procedure. The options that are accepted by the server are forwarded to the open/5 function on the client side. On the client side the open/5 function must accept all option as is or reject the transfer. Then the callback interaction follows the same pattern as described above for the server side. When the last block is encountered in read/1 or write/2 the returned state is forwarded to the user and returned from read_file/3 or write_file/3.

Exports

`prepare(Peer, Access, Filename, Mode, SuggestedOptions, InitialState) -> {ok, AcceptedOptions, NewState} | {error, {Code, Text}}`

Types:

- Peer = {PeerType, PeerHost, PeerPort}
- PeerType = inet | inet6
- PeerHost = ip_address()
- PeerPort = integer()
- Access = read | write
- Filename = string()
- Mode = string()
- SuggestedOptions = AcceptedOptions = [{Key, Value}]
- Key = Value = string()
- InitialState = [] | [{root_dir, string()}]
- NewState = term()
- Code = undef | enoent | eacces | enospc
- | badop | eexist | baduser | badopt
- | int()
- Text = string()

Prepares to open a file on the client side.

No new options may be added, but the ones that are present in `SuggestedOptions` may be omitted or replaced with new values in `AcceptedOptions`.

Will be followed by a call to `open/4` before any read/write access is performed. `AcceptedOptions` is sent to the server which replies with those options that it accepts. These will be forwarded to `open/4` as `SuggestedOptions`.

`open(Peer, Access, Filename, Mode, SuggestedOptions, State) -> {ok, AcceptedOptions, NewState} | {error, {Code, Text}}`

Types:

- Peer = {PeerType, PeerHost, PeerPort}
- PeerType = inet | inet6
- PeerHost = ip_address()
- PeerPort = integer()
- Access = read | write
- Filename = string()
- Mode = string()
- SuggestedOptions = AcceptedOptions = [{Key, Value}]
- Key = Value = string()
- State = InitialState | term()
- InitialState = [] | [{root_dir, string()}]
- NewState = term()
- Code = undef | enoent | eacces | enospc
- | badop | eexist | baduser | badopt
- | int()

- Text = string()

Opens a file for read or write access.

On the client side where the `open/5` call has been preceeded by a call to `prepare/5`, all options must be accepted or rejected.

On the server side, where there is no preceeding `prepare/5` call, no new options may be added, but the ones that are present in `SuggestedOptions` may be omitted or replaced with new values in `AcceptedOptions`.

```
read(State) -> {more, Bin, NewState} | {last, Bin, FileSize} | {error, {Code, Text}}
```

Types:

- State = NewState = term()
- Bin = binary()
- FileSize = int()
- Code = undef | enoent | eacces | enospc
- | badop | eexist | baduser | badopt
- | int()
- Text = string()

Read a chunk from the file.

The callback function is expected to close the file when the last file chunk is encountered. When an error is encountered the callback function is expected to clean up after the aborted file transfer, such as closing open file descriptors etc. In both cases there will be no more calls to any of the callback functions.

```
write(Bin, State) -> {more, NewState} | {last, FileSize} | {error, {Code, Text}}
```

Types:

- Bin = binary()
- State = NewState = term()
- FileSize = int()
- Code = undef | enoent | eacces | enospc
- | badop | eexist | baduser | badopt
- | int()
- Text = string()

Write a chunk to the file.

The callback function is expected to close the file when the last file chunk is encountered. When an error is encountered the callback function is expected to clean up after the aborted file transfer, such as closing open file descriptors etc. In both cases there will be no more calls to any of the callback functions.

```
abort(Code, Text, State) -> ok
```

Types:

- Code = undef | enoent | eacces | enospc
- | badop | eexist | baduser | badopt
- | int()
- Text = string()
- State = term()

Invoked when the file transfer is aborted.

The callback function is expected to clean up its used resources after the aborted file transfer, such as closing open file descriptors etc. The function will not be invoked if any of the other callback functions returns an error, as it is expected that they already have cleaned up the necessary resources. It will however be invoked if the functions fails (crashes).

Glossary

HTTP

Hypertext Transfer Protocol.

RFC

A ”Request for Comments” used as a proposed standard by IETF.

Index of Modules and Functions

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