

inets

Copyright © 1997-2018 Ericsson AB. All Rights Reserved.
inets 6.5
April 27, 2018

Copyright © 1997-2018 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

April 27, 2018

1 Inets User's Guide

The `Inets` application provides a set of Internet-related services as follows:

- An FTP client
- A TFTP client and server
- An client and server

The HTTP client and server are HTTP 1.1 compliant as defined in **RFC 2616**.

1.1 Introduction

1.1.1 Purpose

`Inets` is a container for Internet clients and servers including the following:

- An FTP client
- A TFTP client and server
- An client and server

The HTTP client and server are 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 FTP, TFTP, and HTTP protocols.

1.2 Inets

1.2.1 Service Concept

Each client and server in `Inets` is viewed as a service. Services can be configured to be started at application startup or dynamically in runtime. To run `Inets` as a distributed application that handles application failover and takeover, configure the services to be started at application startup. When starting the `Inets` application, the `Inets` top supervisor starts a number of subsupervisors and worker processes for handling the provided services. When starting services dynamically, new children are added to the supervision tree, unless the service is started with the standalone option. In this case the service is linked to the calling process and all OTP application features, such as soft upgrade, are lost.

Services to be configured for startup at application startup are to be put into the Erlang node configuration file on the following form:

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

For details of what to put in the list of configured services, see the documentation for the services to be configured.

1.3 FTP Client

1.3.1 Getting Started

FTP clients are considered to be rather temporary. Thus, they are only started and stopped during runtime and cannot be started at application startup. The FTP client API is designed to allow some functions to return intermediate results. This implies that only the process that started the FTP client can access it with preserved sane semantics. If the process that started the FTP session dies, the FTP client process terminates.

The client supports IPv6 as long as the underlying mechanisms also do so.

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`:

```
1> inets:start().
ok
2> {ok, Pid} = inets:start(ftpc, [{host, "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> inets:stop(ftpc, Pid).
ok
```

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`.

1.4 HTTP Client

1.4.1 Configuration

The HTTP client default profile is started when the `Inets` application is started and is then available to all processes on that Erlang node. Other profiles can also be started at application startup, or profiles can be started and stopped dynamically in runtime. Each client profile spawns a new process to handle each request, unless a persistent connection can be used with or without pipelining. The client adds 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.

The following is to be put in the Erlang node application configuration file to start a profile at application startup:

```
[{inets, [{services, [{httpc, PropertyList}]}]}]
```

For valid properties, see *httpc(3)*.

1.4.2 Getting Started

Start `Inets`:

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

The following calls use the default client profile. Use the proxy `"www-proxy.mycompany.com:8000"`, except from requests to localhost. This applies to all the following requests.

Example:

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

The following is an ordinary synchronous request:

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

With all the default values presented, a get request can also be written as follows:

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

The following is an ordinary asynchronous request:

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

The result is sent to the calling process as `{http, {ReqestId, Result}}`.

In this case, the calling process is the shell, so the following result is received:

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

This sends a request with a specified connection header:

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

This sends an HTTP request over a unix domain socket (experimental):

```
     8 > httpc:set_options([{ipfamily, local},
     {unix_socket,"/tmp/unix_socket/consul_http.sock"}]).
     9 > {ok, {{NewVersion, 200, NewReasonPhrase}, NewHeaders, NewBody}} =
     httpc:request(put, {"http:///v1/kv/foo", [], [], "hello"}, [], []).
```

Start an HTTP client profile:

```
     10 > {ok, Pid} = inets:start(httpc, [{profile, foo}]).
     {ok, <0.45.0>}
```

The new profile has no proxy settings, so the connection is refused:

```
     11 > httpc:request("http://www.erlang.org", foo).
     {error, econnrefused}
```

Stop the HTTP client profile:

```
12 > inets:stop(httpc, foo).
ok
```

Alternative way to stop the HTTP client profile:

```
13 > inets:stop(httpc, Pid).
ok
```

1.5 HTTP server

1.5.1 Configuration

The HTTP server, also referred to as httpd, handles HTTP requests as described in **RFC 2616** with a few exceptions, such as gateway and proxy functionality. The server supports IPv6 as long as the underlying mechanisms also do so.

The server implements numerous features, such as:

- Secure Sockets Layer (SSL)
- Erlang Scripting Interface (ESI)
- Common Gateway Interface (CGI)
- 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

The configuration of the server is provided as an Erlang property list. For backwards compatibility, a configuration file using apache-style configuration directives is supported.

As of `Inets` 5.0 the HTTP server is an easy to start/stop and customize web server providing the most basic web server functionality. Inets is designed for embedded systems and if you want a full-fledged web server there are exists other erlang open source alternatives.

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

The following is to be put in the Erlang node application configuration file to start an HTTP server at application startup:

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

The server is configured using an Erlang property list. For the available properties, see *httpd(3)*. For backwards compatibility, apache-like configuration files are also supported.

The available configuration properties are as follows:

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

Here:

`{file, file()}`

If you use an old apace-like configuration file.

`{proplist_file, file()}`

File containing an Erlang property list, followed by a full stop, describing the HTTP server configuration.

`{debug, debug()}`

Can enable trace on all functions or only exported functions on chosen modules.

`{accept_timeout, integer()}`

Sets the wanted time-out value for the server to set up a request connection.

1.5.2 Getting Started

Start `Inets`:

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

Start an HTTP server with minimal required configuration. If you specify port `0`, an arbitrary available port is used, and you can use function `info` to find which port number that was picked:

```
2 > {ok, Pid} = inets:start(httpd, [{port, 0},
{server_name,"httpd_test"}, {server_root,"/tmp"},
{document_root,"/tmp/htdocs"}, {bind_address, "localhost"}]).
{ok, 0.79.0}
```

Call `info`:

```
3 >  httpd:info(Pid).
[{mime_types,[{"html","text/html"},{"htm","text/html"}]},
{server_name,"httpd_test"},
{bind_address, {127,0,0,1}},
{server_root,"/tmp"},
{port,59408},
{document_root,"/tmp/htdocs"}]
```

Reload the configuration without restarting the server:

```
4 > httpd:reload_config([{port, 59408},
  {server_name,"httpd_test"}, {server_root,"/tmp/www_test"},
  {document_root,"/tmp/www_test/htdocs"},
  {bind_address, "localhost"}], non_disturbing).
ok.
```

Note:

`port` and `bind_address` cannot be changed. Clients trying to access the server during the reload get a service temporary unavailable answer.

```
5 >  httpd:info(Pid, [server_root, document_root]).
[{server_root,"/tmp/www_test"},{document_root,"/tmp/www_test/htdocs"}]
```

```
6 > ok = inets:stop(httpd, Pid).
```

Alternative:

```
6 > ok = inets:stop(httpd, {{127,0,0,1}, 59408}).
```

Notice that `bind_address` must be the IP address reported by function `info` and cannot be the hostname that is allowed when putting in `bind_address`.

1.5.3 Htaccess - User Configurable Authentication

Web server users without server administrative privileges that need to manage authentication of web pages that are local to their user can use the per-directory runtime configurable user-authentication scheme `htaccess`. It works as follows:

- Each directory in the path to the requested asset is searched for an access file (default is `.htaccess`), which restricts the web servers 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 apply to files in the same directory 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 is applied.
- To change the rules that restrict the use of an asset, the user only needs write access to the directory where the asset is.
- All access files in the path to a requested asset are read once per request. This means that the load on the server increases when `htaccess` is used.
- If a directory is limited both by authentication directives in the HTTP server configuration file and by the `htaccess` files, the user must be allowed to get access to the file by both methods for the request to succeed.

Access Files Directives

In every directory under `DocumentRoot` or under an `Alias` a user can place an access file. An access file is a plain text file that specifies the restrictions to consider before the web server answers 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 is used.

"allow"

Syntax: `Allow` from subnet `subnet | from all`

Default: `from all`

Same as directive `allow` for the server configuration file.

"AllowOverRide"

Syntax: `AllowOverRide all | none | Directives`

Default: `none`

`AllowOverRide` specifies the parameters that access files in subdirectories are not allowed to alter the value for. If the parameter is set to `none`, no further access files is parsed.

If only one access file exists, setting this parameter to `none` can ease the burden on the server as the server then stops looking for access files.

"AuthGroupfile"

Syntax: `AuthGroupFile` Filename

Default: `none`

`AuthGroupFile` indicates which file that contains the list of groups. The 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
```

"AuthName"

Syntax: `AuthName` auth-domain

Default: `none`

Same as directive `AuthName` for the server configuration file.

"AuthType"

Syntax: `AuthType Basic`

Default: `Basic`

`AuthType` specifies which authentication scheme to be used. Only Basic Authenticating using UUEncoding of the password and user ID is implemented.

"AuthUserFile"

Syntax: `AuthUserFile` Filename

Default:`none`

`AuthUserFile` indicates which file that contains the list of users. The filename must contain the absolute path to the file. The username and password are not encrypted so do not place the file with users in a directory that is accessible through the web server. The format of the file is one user per row. Every row contains `UserName` and `Password` separated by a colon, for example:

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

"deny"

Syntax: `deny` from subnet `subnet | from all`

Context: Limit

Same as directive `deny` for the server configuration file.

"Limit"

Syntax: `<Limit` RequestMethods`>`

Default: `none`

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

Example:

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

"order"

Syntax: `order allow deny | deny allow`

Default: `allow deny`

`order` defines if the deny or allow control is to be performed first.

If the order is set to `allow deny`, the users network address is first controlled to be in the allow subset. If the user network address is not in the allowed subset, the user is denied to get the asset. If the network address is in the allowed subset, a second control is performed. That is, the user network address is not in the subset of network addresses to be denied as specified by parameter `deny`.

If the order is set to `deny allow`, only users from networks specified to be in the allowed subset succeeds to request assets in the limited area.

"require"

Syntax: `require group group1 group2... | user user1 user2...`

Default: `none`

Context: Limit

For more information, see directive `require` in *mod_auth(3)*.

1.5.4 Dynamic Web Pages

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

CGI scripts

Common Gateway Interface (CGI) scripts can be written in any programming language. CGI scripts are standardized and supported by most web servers. 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.

ESI-functions

Erlang Server Interface (ESI) functions provide a tight and efficient interface to the execution of Erlang functions. This interface, on the other hand, is `Inets` specific.

CGI Version 1.1, RFC 3875

The module `mod_cgi` enables execution of **CGI scripts** on the server. A file matching the definition of a ScriptAlias config directive is treated as a CGI script. A CGI script is executed by the server and its 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 can be empty.

Example:

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

The server interprets the message headers and most of them are transformed into HTTP headers and sent back to the client together with the message-body.

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

ESI

The Erlang server interface is implemented by module `mod_esi`.

ERL Scheme

The erl scheme is designed to mimic plain CGI, but without the extra overhead. An URL that calls an Erlang `erl` function has the following syntax (regular expression):

```
http://your.server.org/***/Module[:/]Function(?QueryString|/PathInfo)
```

*** 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 function with arity three, as it permits to send chunks of the web page 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 kept for backwards compatibility reasons. For implementation details of the ESI callback function, see *mod_esi(3)*.

EVAL Scheme

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

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

*** 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 take the form as specified in the CGI specification. For implementation details of the ESI callback function, see *mod_esi(3)*.

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,[])))
```

This effectively closes down the Erlang node. Therefore, use the erl scheme instead, until this security breach is fixed.

Today there are no good ways of solving this problem and therefore the eval scheme can be removed in future release of `Inets`.

1.5.5 Logging

Three types of logs are 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

Here:

remotehost
: Remote hostname.

rfc931
The client remote username (**RFC 931**).
authuser
The username used for authentication.
[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 recorded in the error log file. The format of this file is a more unplanned format than the logs using Common Logfile Format, but conforms to the following syntax:

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

1.5.6 Erlang Web Server API

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

- Setting up connections, sending and receiving data.
- URI to filename translation.
- Authentication/access checks.
- Retrieving/generating the response.
- Logging.

To provide customization and extensibility of the request handling of the HTTP servers, most of these steps are handled by one or more modules. These modules can be replaced or removed at runtime and new ones can be added. For each request, all modules are traversed in the order specified by the module directive in the server configuration file. Some parts, mainly the communication- related steps, are considered server core functionality and are not implemented using the Erlang web server API. A description of functionality implemented by the Erlang webserver API is described in *Section Inets Web Server 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 Web Server API modules. This is possible owing to an internal list of key-value tuples, referred to as interaction data.

> **Note:**
>
> Interaction data enforces module dependencies and is to be avoided if possible. This means that the order of modules in the modules property is significant.

API Description

Each module that implements server functionality using the Erlang web server API is to implement the following call back functions:

- `do/1` (mandatory) - the function called when a request is to 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)*.

1.5.7 Inets Web Server Modules

The convention is that all modules implementing some web server functionality has the name `mod_*`. When configuring the web server, an appropriate selection of these modules is to be present in the module directive. Notice that there are some interaction dependencies to take into account, so the order of the modules cannot be random.

mod_action - Filetype/Method-Based Script Execution

This module runs CGI scripts whenever a file of a certain type or HTTP method (see **RFC 1945**) is requested.

Uses the following Erlang Web Server API interaction data:

- `real_name` - from *mod_alias*.

Exports the following Erlang Web Server API interaction data, if possible:

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

mod_alias - URL Aliasing

The *mod_alias* module makes it possible to map different parts of the host file system into the document tree, that is, creates aliases and redirections.

Exports the following Erlang Web Server API interaction data, if possible:

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

mod_auth - User Authentication

The *mod_auth(3)* module provides for basic user authentication using textual files, Dets databases as well as Mnesia databases.

Uses the following Erlang Web Server API interaction data:

- `real_name` - from *mod_alias*

Exports the following Erlang Web Server API interaction data:

`{remote_user, User}`
The username used for authentication.

Mnesia As Authentication Database

If Mnesia is used as storage method, Mnesia must be started before the HTTP server. The first time Mnesia is started, the schema and the tables must be created before Mnesia is started. A simple example of a module with two functions that creates and start Mnesia is provided here. Function `first_start/0` is to be used the first time. It creates the schema and the tables. `start/0` is to be used in consecutive startups. `start/0` starts Mnesia and waits for the tables to be initiated. This function must only be used when the schema and the tables are already created.

```
-module(mnesia_test).
-export([start/0,load_data/0]).
-include_lib("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 that file must be included. `first_start/0` creates a schema that specifies on which nodes the database is to 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 of how to create the table, see *`mnesia(3)`*, documentation for more information. As the implementation of the `mod_auth_mnesia` saves one row for each user, the type must be `bag`. When the schema and the tables are created, function *mnesia:start/0* is used to start Mnesia and waits for the tables to be loaded. Mnesia uses the directory specified as `mnesia_dir` at startup if specified, otherwise Mnesia uses the current directory. For security reasons, ensure that the Mnesia tables are stored outside the document tree of the HTTP server. If they are placed in the directory which it protects, clients can download the tables. Only the Dets and Mnesia storage methods allow writing of dynamic user data to disk. `plain` is a read only method.

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 must be removed from the Modules config directive if directory listings is unwanted.

Uses the following Erlang Web Server API interaction data:

- `real_name` - from *mod_alias*

Exports the following Erlang Web Server API interaction data:

`{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 Web Server API interaction data:

- `remote_user` - from `mod_auth`

mod_esi - Erlang Server Interface

The *mod_esi(3)* module implements the Erlang Server Interface (ESI) providing a tight and efficient interface to the execution of Erlang functions.

Uses the following Erlang web server API interaction data:

- `remote_user` - from `mod_auth`

Exports the following Erlang web server API interaction data:

`{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 web server API interaction data:

- `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 Web Server API interaction data:

- `real_name` - from *mod_alias*

mod_htaccess - User Configurable Access

This module provides per-directory user configurable access control.

Uses the following Erlang Web Server API interaction data:

- `real_name` - from *mod_alias*

Exports the following Erlang Web Server API interaction data:

`{remote_user_name, User}`
The username used for authentication.

mod_log - Logging Using Text Files.

Standard logging using the "Common Logfile Format" and text files.

Uses the following Erlang Web Server API interaction data:

- `remote_user` - from `mod_auth`

mod_range - Requests with Range Headers

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

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

Uses the following Erlang Web Server API interaction data:

- `real_name` - from *mod_alias*

mod_response_control - Requests with If* Headers

This module controls that the conditions in the requests are fulfilled. For example, a request can specify that the answer only is of interest if the content is unchanged since the last retrieval. If the content is changed, the range request is to be converted to a request for the whole file instead.

If a client sends more than one of the header fields that restricts the servers right to respond, the standard does not specify how this is to be handled. *httpd(3)* controls each field in the following order and if one of the fields does not match the current state, the request is rejected with a proper response:

`If-modified`

`If-Unmodified`

`If-Match`

`If-Nomatch`

Uses the following Erlang Web Server API interaction data:

- `real_name` - from *mod_alias*

Exports the following Erlang Web Server API interaction data:

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

mod_security - Security Filter

The *mod_security* module serves as a filter for authenticated requests handled in *mod_auth(3)*. It provides a 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 calls a configurable callback module when the events occur.

There is also an API to block or unblock users manually. This API can also 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.

2 Reference Manual

`Inets` is a container for Internet clients and servers. An FTP client, an HTTP client and server, and a TFTP client and server are incorporated in `Inets`.

inets

Erlang module

This module provides the most basic API to the clients and servers that are part of the `Inets` application, such as start and stop.

DATA TYPES

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

```
service() = ftpc | tftp | httpc | httpd
property() = atom()
```

Exports

`services() -> [{Service, Pid}]`

Types:

`Service = service()`
`Pid = pid()`

Returns a list of currently running services.

> **Note:**
>
> Services started as `stand_alone` are not listed.

`services_info() -> [{Service, Pid, Info}]`

Types:

`Service = service()`
`Pid = pid()`
`Info = [{Option, Value}]`
`Option = property()`
`Value = term()`

Returns a list of currently running services where each service is described by an `[{Option, Value}]` list. The information in the list is specific for each service and each service has probably its own info function that gives more details about the service.

`service_names() -> [Service]`

Types:

`Service = service()`

Returns a list of available service names.

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

Types:

Type = permanent | transient | temporary

Starts the `Inets` application. Default type is `temporary`. See also *application(3)*.

start(Service, ServiceConfig) -> {ok, Pid} | {error, Reason}
start(Service, ServiceConfig, How) -> {ok, Pid} | {error, Reason}

Types:

Service = service()
ServiceConfig = [{Option, Value}]
Option = property()
Value = term()
How = inets | stand_alone - default is inets.

Dynamically starts an `Inets` service after the `Inets` application has been started.

> **Note:**
>
> Dynamically started services are not handled by application takeover and failover behavior when `Inets` is run as a distributed application. Nor are they automatically restarted when the `Inets` application is restarted. As long as the `Inets` application is operational, they are supervised and can be soft code upgraded.
>
> A service started as `stand_alone`, that is, the service is not started as part of the `Inets` application, lose all OTP application benefits, such as soft upgrade. The `stand_alone`-service is linked to the process that started it. Usually some supervision functionality is still in place and in some sense the calling process becomes the top supervisor.

stop() -> ok

Stops the `Inets` application. See also *application(3)*.

stop(Service, Reference) -> ok | {error, Reason}

Types:

Service = service() | stand_alone
Reference = pid() | term() - service-specified reference
Reason = term()

Stops a started service of the `Inets` application or takes down a `stand_alone`-service gracefully. When option `stand_alone` is used in start, only the pid is a valid argument to stop.

SEE ALSO

ftp(3), httpc(3), httpd(3), tftp(3)

ftp

Erlang module

This module implements a client for file transfer according to a subset of the File Transfer Protocol (FTP), see **RFC 959**.

As from `Inets` 4.4.1, the FTP client always tries to use passive FTP mode and only resort to active FTP mode if this fails. This default behavior can be changed by start option *mode*.

An FTP client can be started in two ways. One is using the *Inets service framework*, the other is to start it directly as a standalone process using function *open*.

For a simple example of an FTP session, see *Inets User's Guide*.

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 be stored as remote files (see `send_bin/3`).

A set of functions is provvided 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`. For receive, see `recv_chunk_start/2` and `recv_chunk/`).

The return values of the following functions depend 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 example, 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 a directory that exists but is empty.

FTP CLIENT SERVICE START/STOP

The FTP client can be started and stopped dynamically in runtime by calling the `Inets` application API `inets:start(ftpc, ServiceConfig)`, or `inets:start(ftpc, ServiceConfig, How)`, and `inets:stop(ftpc, Pid)`. For details, see *inets(3)*.

The available configuration options are as follows:

{host, Host}

Host = `string() | ip_address()`

{port, Port}

Port = `integer() > 0`

Default is `21`.

{mode, Mode}

Mode = `active | passive`

Default is `passive`.

{verbose, Verbose}

Verbose = `boolean()`

Determines if the FTP communication is to be verbose or not.

Default is `false`.

{debug, Debug}

Debug = `trace | debug | disable`

Debugging using the dbg toolkit.

Default is `disable`.

{ipfamily, IpFamily}

IpFamily = `inet | inet6 | inet6fb4`

With `inet6fb4` the client behaves as before, that is, tries to use IPv6, and only if that does not work it uses IPv4).

Default is `inet` (IPv4).

{timeout, Timeout}

Timeout = `non_neg_integer()`

Connection time-out.

Default is `60000` (milliseconds).

{dtimeout, DTimeout}

DTimeout = `non_neg_integer() | infinity`

Data connect time-out. The time the client waits for the server to connect to the data socket.

Default is `infinity`.

{progress, Progress}

Progress = `ignore | {CBModule, CBFunction, InitProgress}`

`CBModule = atom()`, `CBFunction = atom()`

`InitProgress = term()`

Default is `ignore`.

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

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

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

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

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

- At the end of the file the following call is made to indicate the end of the transfer:

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

The callback function is to be defined as follows:

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

CBModule = CBFunction = atom()

UserProgressTerm = term()

File = string()

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

For remote files, `ftp` cannot determine the file size in a platform independent way. In this case the size becomes `unknown` and it is left to the application to determine the size.

Note:

The callback is made by a middleman process, hence the file transfer is not affected by the code in the progress callback function. If the callback crashes, this is detected by the FTP connection process, which then prints an info-report and goes on 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 (compare `lpwd/1`) is set to the value reported by `file:get_cwd/1`, the wanted local directory.

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

DATA TYPES

The following type definitions 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` - all restrictions are not always relevant to all functions

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

Exports

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

Types:

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

Sets the account for an operation, if needed.

`append(Pid, LocalFile) ->`
`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 is appended to is set to `RemoteFile`, otherwise to `LocalFile`. If the file does not exists, it is 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 appends it to the file `RemoteFile`. If the file does not exist, it is created.

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

Types:

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

Transfers the chunk `Bin` to the remote server, which appends it to the file specified in the call to `append_chunk_start/2`.

For some errors, for example, file system full, it is necessary 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()

Starts the transfer of chunks for appending to the file `File` at the remote server. If the file does not exist, it is 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 an FTP session, created using function *open*.

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) ->`
`ls(Pid, Pathname) -> {ok, Listing} | {error, Reason}`

Types:

```
Pid = pid()
Pathname = string()
Listing = string()
Reason = restriction_reason() | common_reason()
```

Returns a list of files in long format.

`Pathname` can be a directory, a group of files, or a file. The `Pathname` string can contain wildcards.

`ls/1` implies the current remote directory of the user.

The format of `Listing` depends on the operating system. 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) ->
nlist(Pid, Pathname) -> {ok, Listing} | {error, Reason}
```

Types:

```
Pid = pid()
Pathname = string()
Listing = string()
Reason = restriction_reason() | common_reason()
```

Returns a list of files in short format.

`Pathname` can be a directory, a group of files, or a file. The `Pathname` string can contain wildcards.

`nlist/1` implies the current remote directory of the user.

The format of `Listing` is a stream of filenames where each filename is separated by <CRLF> or <NL>. Contrary to function `ls`, the purpose of `nlist` is to enable a program to process filename information automatically.

```
open(Host) -> {ok, Pid} | {error, Reason}
open(Host, Opts) -> {ok, Pid} | {error, Reason}
```

Types:

```
Host = string() | ip_address()
Opts = options()
options() = [option()]
option() = start_option() | open_option()
start_option() = {verbose, verbose()} | {debug, debug()}
verbose() = boolean() (default is false)
debug() = disable | debug | trace (default is disable)
open_option() = {ipfamily, ipfamily()} | {port, port()} | {mode, mode()}
| {tls, tls_options()} | {timeout, timeout()} | {dtimeout, dtimeout()} |
{progress, progress()}
ipfamily() = inet | inet6 | inet6fb4 (default is inet)
port() = integer() > 0 (default is 21)
mode() = active | passive (default is passive)
tls_options() = [ssl:ssloption()]
timeout() = integer() > 0 (default is 60000 milliseconds)
dtimeout() = integer() > 0 | infinity (default is infinity)
pogress() = ignore | {module(), function(), initial_data()} (default is
ignore)
module() = atom()
function() = atom()
initial_data() = term()
Reason = ehost | term()
```

Starts a standalone FTP client process (without the `Inets` service framework) and opens a session with the FTP server at `Host`.

If option `{tls, tls_options()}` is present, the FTP session is transported over `tls` (`ftps`, see **RFC 4217**). The list `tls_options()` can be empty. The function *`ssl:connect/3`* is used for securing both the control connection and the data sessions.

A session opened in this way is closed using function *close*.

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) ->
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

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

If the file write fails (for example, `enospc`), the command is aborted and `{error, file_write_error_reason()}` is returned. However, the file is **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()

Starts 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()

Receives a chunk of the remote file (`RemoteFile` of `recv_chunk_start`). The return values have 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) ->`
`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 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()
```

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

For some errors, for example, file system full, it is necessary 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()

Starts 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 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 `Password` to the account specified by `Account`.

quote(Pid, Command) -> [FTPLine]

Types:

Pid = pid()
Command = string()
FTPLine = string(

> **Note:**
>
> The telnet end of line characters, from the FTP protocol definition, CRLF, for example, "\\r\\n" has been removed.

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

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

ERRORS

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

`echunk`

Synchronization error during chunk sending according to one of the following:

- A call is made to `send_chunk/2` or `send_chunk_end/1` before a call to `send_chunk_start/2`.
- A call has been made to another transfer function during chunk sending, that is, before a call to `send_chunk_end/1`.

`eclosed`

The session is closed.

`econn`

Connection to the remote server is prematurely closed.

`ehost`

Host is not found, FTP server is not found, or connection is rejected by FTP server.

`elogin`

User is 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`

Invalid username or password.

`etnospc`

Insufficient storage space in system [452].

`epnospc`

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

`efnamena`

Filename not allowed [553].

SEE ALSO

file(3) filename(3) and J. Postel and J. Reynolds: File Transfer Protocol (**RFC 959**).

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 is the "netascii" transfer mode.

The *start/1* function starts a daemon process listening for UDP packets on a port. When it receives a request for read or write, it spawns a temporary server process handling the transfer.

On the client side, function *read_file/3* and *write_file/3* spawn a temporary client process establishing contact with a TFTP daemon and perform the file transfer.

`tftp` uses a callback module to handle the file transfer. Two such callback modules are provided, `tftp_binary` and `tftp_file`. See *read_file/3* and *write_file/3* for details. You can also implement your own callback modules, see *CALLBACK FUNCTIONS*. A callback module provided by the user is registered using option `callback`, see *DATA TYPES*.

TFTP SERVER SERVICE START/STOP

A TFTP server can be configured to start statically when starting the `Inets` application. Alternatively, it can be started dynamically (when `Inets` is already started) by calling the `Inets` application API `inets:start(tftpd, ServiceConfig)` or `inets:start(tftpd, ServiceConfig, How)`, see *inets(3)* for details. The `ServiceConfig` for TFTP is described in the *DATA TYPES* section.

The TFTP server can be stopped using `inets:stop(tftpd, Pid)`, see *inets(3)* for details.

The TPFT client is of such a temporary nature that it is not handled as a service in the `Inets` service framework.

DATA TYPES

```
ServiceConfig = Options

Options = [option()]
```

Most of the options are common for both the client and the server side, but some of them differs a little. The available `option()`s are as follows:

`{debug, Level}`

`Level = none | error | warning | brief | normal | verbose | all`

Controls the level of debug printouts. 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. Defaults is the standardized number 69. On the server side, it can sometimes make sense to set it to 0, meaning that the daemon just picks a free port (which one is returned by function `info/1`).

If a socket is connected already, option `{udp, [{fd, integer()}]}` can be used to pass the open file descriptor to `gen_udp`. This can be automated by using a command-line argument stating the prebound file descriptor number. For example, if the port is 69 and file descriptor 22 is opened by `setuid_socket_wrap`, the command-line argument "-tftpd_69 22" triggers the prebound file descriptor 22 to be used instead of opening port 69. The UDP option `{udp, [{fd, 22}]}` is automatically 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 that is used by the server/client during the file transfer. Default is `random`, which is the standardized policy. With this policy a randomized free port is used. A single port or a range of ports can be useful if the protocol passes through a firewall.

`{udp, Options}`

`Options = [Opt]`, see *gen_udp:open/2*.

`{use_tsize, Bool}`

```
Bool = bool()
```

Flag for automated use of option `tsize`. With this set to `true`, the `write_file/3` client determines the filesize and sends it to the server as the standardized `tsize` option. A `read_file/3` client acquires only a filesize from the server by sending a zero `tsize`.

`{max_tsize, MaxTsize}`

```
MaxTsize = int() | infinity
```

Threshold for the maximal filesize in bytes. The transfer is aborted if the limit is exceeded. Default is `infinity`.

`{max_conn, MaxConn}`

```
MaxConn = int() | infinity
```

Threshold for the maximal number of active connections. The daemon rejects the setup of new connections if the limit is exceeded. Default is `infinity`.

`{TftpKey, TftpVal}`

```
TftpKey = string()
TftpVal = string()
```

Name and value of a TFTP option.

`{reject, Feature}`

```
Feature = Mode | TftpKey
 Mode = read | write
 TftpKey = string()
```

Controls which features to reject. This is mostly useful for the server as it can restrict the use 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 is matched to the regular expressions of the registered callbacks. The first matching callback is used during the transfer. See *read_file/3* and *write_file/3*.

The callback module must implement the `tftp` behavior, see *CALLBACK FUNCTIONS*.

`{logger, Module}`

`Module = module()()`

Callback module for customized logging of errors, warnings, and info messages. The callback module must implement the `tftp_logger` behavior, see *LOGGER FUNCTIONS*. The default module is `tftp_logger`.

`{max_retries, MaxRetries}`

`MaxRetries = int()`

Threshold for the maximal number of retries. By default the server/client tries to resend a message up to five times when the time-out expires.

Exports

`change_config(daemons, Options) -> [{Pid, Result}]`

Types:

`Options = [option()]`
`Pid = pid()`
`Result = ok | {error, Reason}`
`Reason = term()`

Changes configuration for all TFTP daemon processes.

`change_config(servers, Options) -> [{Pid, Result}]`

Types:

`Options = [option()]`
`Pid = pid()`
`Result = ok | {error, Reason}`
`Reason = term()`

Changes configuration for all TFTP server processes.

`change_config(Pid, Options) -> Result`

Types:

`Pid = pid()`
`Options = [option()]`
`Result = ok | {error, Reason}`
`Reason = term()`

Changes configuration for a TFTP daemon, server, or client process.

`info(daemons) -> [{Pid, Options}]`

Types:

`Pid = [pid()()]`

```
Options = [option()]
Reason = term()
```

Returns information about all TFTP daemon processes.

info(servers) -> [{Pid, Options}]

Types:

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

Returns information about all TFTP server processes.

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

Types:

```
Options = [option()]
Reason = term()
```

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

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.

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

Types:

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

Starts a daemon process listening for UDP packets on a port. When it receives a request for read or write, it spawns a temporary server process handling the actual transfer of the (virtual) file.

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.

CALLBACK FUNCTIONS

A `tftp` callback module is to be implemented as a `tftp` behavior and export the functions listed in the following.

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 function `read/1` or `write/2` is invoked repeatedly, once per transferred block. At each function call, the state returned from the previous call is obtained. When the last block is encountered, function `read/1` or `write/2` is expected to close the (virtual) file and return its last state. Function `abort/3` is only used in error situations. Function `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 to return the subset of them that it accepts. Then the options are sent to the server, which performs the same TFTP option negotiation procedure. The options that are accepted by the server are forwarded to function `open/5` on the client side. On the client side, function `open/5` must accept all option as-is or reject the transfer. Then the callback interaction follows the same pattern as described 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`.

If a callback (performing the file access in the TFTP server) takes too long time (more than the double TFTP time-out), the server aborts the connection and sends an error reply to the client. This implies that the server releases resources attached to the connection faster than before. The server simply assumes that the client has given up.

If the TFTP server receives yet another request from the same client (same host and port) while it already has an active connection to the client, it ignores the new request if the request is equal to the first one (same filename and options). This implies that the (new) client will be served by the already ongoing connection on the server side. By not setting up yet another connection, in parallel with the ongoing one, the server consumes less resources.

Exports

Module: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 and so on. The function is not invoked if any of the other callback functions returns an error, as it is expected that they already have cleaned up the necessary resources. However, it is invoked if the functions fail (crash).

`Module: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 preceded by a call to `prepare/5`, all options must be accepted or rejected.

On the server side, where there is no preceding `prepare/5` call, no new options can be added, but those present in `SuggestedOptions` can be omitted or replaced with new values in `AcceptedOptions`.

`Module: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 can be added, but those present in `SuggestedOptions` can be omitted or replaced with new values in `AcceptedOptions`.

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

`Module: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()
```

Reads 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, and so on. In both cases there will be no more calls to any of the callback functions.

`Module: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()
```

Writes 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, and so on. In both cases there will be no more calls to any of the callback functions.

LOGGER FUNCTIONS

A `tftp_logger` callback module is to be implemented as a `tftp_logger` behavior and export the following functions:

Exports

`Logger:error_msg(Format, Data) -> ok | exit(Reason)`

Types:

`Format = string()`
`Data = [term()]`
`Reason = term()`

Logs an error message. See `error_logger:error_msg/2` for details.

`Logger:info_msg(Format, Data) -> ok | exit(Reason)`

Types:

`Format = string()`
`Data = [term()]`
`Reason = term()`

Logs an info message. See `error_logger:info_msg/2` for details.

`Logger:warning_msg(Format, Data) -> ok | exit(Reason)`

Types:

`Format = string()`
`Data = [term()]`
`Reason = term()`

Logs a warning message. See `error_logger:warning_msg/2` for details.

httpc

Erlang module

This module provides the API to an HTTP/1.1 compatible client according to **RFC 2616**. Caching is not supported.

Note:

When starting the `Inets` application, a manager process for the default profile is started. The functions in this API that do not explicitly use a profile accesses the default profile. A profile keeps track of proxy options, cookies, and other options that can be applied to more than one request.

If the scheme `https` is used, the `SSL` application must be started. When `https` links need to go through a proxy, the CONNECT method extension to HTTP-1.1 is used to establish a tunnel and then the connection is upgraded to TLS. However, "TLS upgrade" according to **RFC 2817**is not supported.

Pipelining is only used if the pipeline time-out is set, otherwise persistent connections without pipelining are used. That is, the client always waits for the previous response before sending the next request.

Some examples are provided in the *Inets User's Guide*.

DATA TYPES

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

`boolean() = true | false`

`string()` = list of ASCII characters

`request_id() = reference()`

`profile() = atom()`

`path() = string()` representing a file path or directory path

`ip_address()` = See the *inet(3)* manual page in Kernel.

`socket_opt()` = See the options used by *gen_tcp(3)* `gen_tcp(3)` and *ssl(3)* connect(s)

HTTP DATA TYPES

Type definitions related to HTTP:

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

`request()`

= `{url(), headers()}`

| `{url(), headers(), content_type(), body()}`

`url()` = `string()` syntax according to the URI definition in **RFC 2396**, for example `"http://www.erlang.org"`

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

`http_version() = string()`, for example, `"HTTP/1.1"`

`status_code() = integer()`

`reason_phrase() = string()`

`content_type() = string()`

```
headers() = [header()]
header() = {field(), value()}
field() = string()
value() = string()
body()
    = string() | binary()
    | {fun(accumulator())
    -> body_processing_result(), accumulator()}
    | {chunkify, fun(accumulator())
    -> body_processing_result(), accumulator()}
body_processing_result() = eof | {ok, iolist(), accumulator()}
accumulator() = term()
filename() = string()
```

For more information about HTTP, see **RFC 2616**.

SSL DATA TYPES

See *ssl(3)* for information about SSL options (`ssloptions()`).

HTTP CLIENT SERVICE START/STOP

An HTTP client can be configured to start when starting the `Inets` application or started dynamically in runtime by calling the `Inets` application API `inets:start(httpc, ServiceConfig)` or `inets:start(httpc, ServiceConfig, How)`, see *inets(3)*. The configuration options are as follows:

{profile, profile()}

Name of the profile, see *DATA TYPES*. This option is mandatory.

{data_dir, path()}

Directory where the profile can save persistent data. If omitted, all cookies are treated as session cookies.

The client can be stopped using `inets:stop(httpc, Pid)` or `inets:stop(httpc, Profile)`.

Exports

`cancel_request(RequestId) ->`
`cancel_request(RequestId, Profile) -> ok`

Types:

`RequestId = request_id() - A unique identifier as returned by request/4`
`Profile = profile() | pid()`

When started `stand_alone` only the pid can be used.

Cancels an asynchronous HTTP request. Notice that this does not guarantee that the request response is not delivered. Because it is asynchronous, the request can already have been completed when the cancellation arrives.

cookie_header(Url) ->
cookie_header(Url, Profile | Opts) -> header() | {error, Reason}
cookie_header(Url, Opts, Profile) -> header() | {error, Reason}

Types:

Url = url()

Opts = [cookie_header_opt()]

Profile = profile() | pid()

When started `stand_alone`.

cookie_header_opt() = {ipv6_host_with_brackets, boolean()}

Returns the cookie header that would have been sent when making a request to `Url` using profile `Profile`. If no profile is specified, the default profile is used.

Option `ipv6_host_with_bracket` deals with how to parse IPv6 addresses. For details, see argument `Options` of *request/[4,5]*.

get_options(OptionItems) -> {ok, Values} | {error, Reason}
get_options(OptionItems, Profile) -> {ok, Values} | {error, Reason}

Types:

OptionItems = all | [option_item()]

option_item() = proxy | https_proxy | max_sessions | keep_alive_timeout | max_keep_alive_length | pipeline_timeout | max_pipeline_length | cookies | ipfamily | ip | port | socket_opts | verbose | unix_socket

Profile = profile() | pid()

When started `stand_alone` only the pid can used.

Values = [{option_item(), term()}]

Reason = term()

Retrieves the options currently used by the client.

info() -> list()
info(Profile) -> list()

Types:

Profile = profile() | pid()

When started `stand_alone` only the pid can be used.

Produces a list of miscellaneous information. Intended for debugging. If no profile is specified, the default profile is used.

reset_cookies() -> void()
reset_cookies(Profile) -> void()

Types:

Profile = profile() | pid()

When started `stand_alone` only the pid can be used.

Resets (clears) the cookie database for the specified `Profile`. If no profile is specified the default profile is used.

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

Types:

Url = url()

Result = {status_line(), headers(), Body} | {status_code(), Body} | request_id()

Body = string() | binary()

Profile = profile() | pid()

When started `stand_alone` only the pid can be used.

Reason = term()

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

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

Types:

Method = method()

Request = request()

HTTPOptions = http_options()

http_options() = [http_option()]

http_option() = {timeout, timeout()} | {connect_timeout, timeout()} | {ssl, ssloptions()} | {essl, ssloptions()} | {autoredirect, boolean()} | {proxy_auth, {userstring(), passwordstring()}} | {version, http_version()} | {relaxed, boolean()} | {url_encode, boolean()}

timeout() = integer() >= 0 | infinity

Options = options()

options() = [option()]

option() = {sync, boolean()} | {stream, stream_to()} | {body_format, body_format()} | {full_result, boolean()} | {headers_as_is, boolean() | {socket_opts, socket_opts()} | {receiver, receiver()} | {ipv6_host_with_brackets, boolean()}

stream_to() = none | self | {self, once} | filename()

socket_opts() = [socket_opt()]

receiver() = pid() | function()/1 | {Module, Function, Args}

Module = atom()

Function = atom()

Args = list()

body_format() = string | binary

Result = {status_line(), headers(), Body} | {status_code(), Body} | request_id()

Body = string() | binary()

Profile = profile() | pid()

When started `stand_alone` only the pid can be used.

Reason = {connect_failed, term()} | {send_failed, term()} | term()

Sends an HTTP request. The function can be both synchronous and asynchronous. In the latter case, the function returns `{ok, RequestId}` and then the information is delivered to the `receiver` depending on that value.

HTTP option (`http_option()`) details:

`timeout`

Time-out time for the request.

The clock starts ticking when the request is sent.

Time is in milliseconds.

Default is `infinity`.

`connect_timeout`

Connection time-out time, used during the initial request, when the client is **connecting** to the server.

Time is in milliseconds.

Default is the value of option `timeout`.

`ssl`

This is the `SSL/TLS` connectin configuration option.

Defaults to `[]`. See *ssl:connect/[2,3,4]* for available options.

`autoredirect`

The client automatically retrieves the information from the new URI and returns that as the result, instead of a 30X-result code.

For some 30X-result codes, automatic redirect is not allowed. In these cases the 30X-result is always returned.

Default is `true`.

`proxy_auth`

A proxy-authorization header using the provided username and password is added to the request.

`version`

Can be used to make the client act as an `HTTP/1.0` or `HTTP/0.9` client. By default this is an `HTTP/1.1` client. When using `HTTP/1.0` persistent connections are not used.

Default is the string `"HTTP/1.1"`.

`relaxed`

If set to `true`, workarounds for known server deviations from the HTTP-standard are enabled.

Default is `false`.

`url_encode`

Applies Percent-encoding, also known as URL encoding on the URL.

Default is `false`.

Option (`option()`) details:

`sync`

Option for the request to be synchronous or asynchronous.

Default is `true`.

`stream`

Streams the body of a 200 or 206 response to the calling process or to a file. When streaming to the calling process using option `self`, the following stream messages are sent to that process: `{http, {RequestId, stream_start, Headers}}`, `{http, {RequestId, stream, BinBodyPart}}`, and `{http, {RequestId, stream_end, Headers}}`.

When streaming to the calling processes using option `{self, once}`, the first message has an extra element, that is, `{http, {RequestId, stream_start, Headers, Pid}}`. This is the process id to be used as an argument to `httpc:stream_next/1` to trigger the next message to be sent to the calling process.

Notice that chunked encoding can add headers so that there are more headers in the `stream_end` message than in `stream_start`. When streaming to a file and the request is asynchronous, the message `{http, {RequestId, saved_to_file}}` is sent.

Default is `none`.

`body_format`

Defines if the body is to be delivered as a string or binary. This option is only valid for the synchronous request.

Default is `string`.

`full_result`

Defines if a "full result" is to be returned to the caller (that is, the body, the headers, and the entire status line) or not (the body and the status code).

Default is `true`.

`headers_as_is`

Defines if the headers provided by the user are to be made lower case or to be regarded as case sensitive.

The HTTP standard requires them to be case insensitive. Use this feature only if there is no other way to communicate with the server or for testing purpose. When this option is used, no headers are automatically added. All necessary headers must be provided by the user.

Default is `false`.

`socket_opts`

Socket options to be used for this and subsequent requests.

Overrides any value set by function *set_options*.

The validity of the options is **not** checked by the HTTP client they are assumed to be correct and passed on to ssl application and inet driver, which may reject them if they are not correct. Note that the current implementation assumes the requests to the same host, port combination will use the same socket options.

By default the socket options set by function *set_options/[1,2]* are used when establishing a connection.

`receiver`

Defines how the client delivers the result of an asynchronous request (`sync` has the value `false`).

`pid()`

Messages are sent to this process in the format `{http, ReplyInfo}`.

`function/1`

Information is delivered to the receiver through calls to the provided fun `Receiver(ReplyInfo)`.

`{Module, Function, Args}`

Information is delivered to the receiver through calls to the callback function `apply(Module, Function, [ReplyInfo | Args])`.

In all of these cases, `ReplyInfo` has the following structure:

```
{RequestId, saved_to_file}
{RequestId, {error, Reason}}
{RequestId, Result}
{RequestId, stream_start, Headers}
{RequestId, stream_start, Headers, HandlerPid}
{RequestId, stream, BinBodyPart}
{RequestId, stream_end, Headers}
```

Default is the `pid` of the process calling the request function (`self()`).

`ipv6_host_with_brackets`

Defines when parsing the Host-Port part of an URI with an IPv6 address with brackets, if those brackets are to be retained (`true`) or stripped (`false`).

Default is `false`.

`set_options(Options) ->`
`set_options(Options, Profile) -> ok | {error, Reason}`

Types:

`Options = [Option]`
`Option = {proxy, {Proxy, NoProxy}}`
`| {https_proxy, {Proxy, NoProxy}}`
`| {max_sessions, MaxSessions}`
`| {max_keep_alive_length, MaxKeepAlive}`
`| {keep_alive_timeout, KeepAliveTimeout}`
`| {max_pipeline_length, MaxPipeline}`
`| {pipeline_timeout, PipelineTimeout}`
`| {cookies, CookieMode}`
`| {ipfamily, IpFamily}`
`| {ip, IpAddress}`
`| {port, Port}`
`| {socket_opts, socket_opts()}`
`| {verbose, VerboseMode}`
`| {unix_socket, UnixSocket}`
`Proxy = {Hostname, Port}`
`Hostname = string()`

Example: "localhost" or "foo.bar.se"

`Port = integer()`

Example: 8080

`NoProxy = [NoProxyDesc]`
`NoProxyDesc = DomainDesc | HostName | IPDesc`
`DomainDesc = "*.Domain"`

Example: "*.ericsson.se"

`IpDesc = string()`

Example: "134.138" or "[FEDC:BA98" (all IP addresses starting with 134.138 or FEDC:BA98), "66.35.250.150" or "[2010:836B:4179::836B:4179]" (a complete IP address). `proxy` defaults to `{undefined, []}`, that is, no proxy is configured and `https_proxy` defaults to the value of `proxy`.

`MaxSessions = integer()`

Maximum number of persistent connections to a host. Default is `2`.

`MaxKeepAlive = integer()`

Maximum number of outstanding requests on the same connection to a host. Default is `5`.

`KeepAliveTimeout = integer()`

If a persistent connection is idle longer than the `keep_alive_timeout` in milliseconds, the client closes the connection. The server can also have such a time-out but do not take that for granted. Default is `120000` (= 2 min).

`MaxPipeline = integer()`

Maximum number of outstanding requests on a pipelined connection to a host. Default is `2`.

`PipelineTimeout = integer()`

If a persistent connection is idle longer than the `pipeline_timeout` in milliseconds, the client closes the connection. Default is `0`, which results in pipelining not being used.

`CookieMode = enabled | disabled | verify`

If cookies are enabled, all valid cookies are automatically saved in the cookie database of the client manager. If option `verify` is used, function `store_cookies/2` has to be called for the cookies to be saved. Default is `disabled`.

`IpFamily = inet | inet6 | local`

Default is `inet`.

`IpAddress = ip_address()`

If the host has several network interfaces, this option specifies which one to use. See *gen_tcp:connect/3,4* for details.

`Port = integer()`

Local port number to use. See *gen_tcp:connect/3,4* for details.

`socket_opts() = [socket_opt()]`

The options are appended to the socket options used by the client. These are the default values when a new request handler is started (for the initial connect). They are passed directly to the underlying transport (`gen_tcp` or `SSL`) **without** verification.

`VerboseMode = false | verbose | debug | trace`

Default is `false`. This option is used to switch on (or off) different levels of Erlang trace on the client. It is a debug feature.

`Profile = profile() | pid()`

When started `stand_alone` only the pid can be used.

`UnixSocket = path()`

Experimental option for sending HTTP requests over a unix domain socket. The value of `unix_socket` shall be the full path to a unix domain socket file with read/write permissions for the erlang process. Default is `undefined`.

Sets options to be used for subsequent requests.

Note:

If possible, the client keeps its connections alive and uses persistent connections with or without pipeline depending on configuration and current circumstances. The HTTP/1.1 specification does not provide a guideline for how many requests that are ideal to be sent on a persistent connection. This depends much on the application.

A long queue of requests can cause a user-perceived delay, as earlier requests can take a long time to complete. The HTTP/1.1 specification suggests a limit of two persistent connections per server, which is the default value of option `max_sessions`.

```
store_cookies(SetCookieHeaders, Url) ->
store_cookies(SetCookieHeaders, Url, Profile) -> ok | {error, Reason}
```

Types:

```
SetCookieHeaders = headers() - where field = "set-cookie"
Url = url()
Profile = profile() | pid()
```

When started `stand_alone` only the pid can be used.

Saves the cookies defined in `SetCookieHeaders` in the client profile cookie database. Call this function if option `cookies` is set to `verify`. If no profile is specified, the default profile is used.

```
stream_next(Pid) -> ok
```

Types:

```
Pid = pid()
```

As received in the `stream_start message`

Triggers the next message to be streamed, that is, the same behavior as active ones for sockets.

```
which_cookies() -> cookies()
which_cookies(Profile) -> cookies()
```

Types:

```
Profile = profile() | pid()
```

When started `stand_alone` only the pid can be used.

```
cookies() = [cookie_stores()]
cookie_stores() = {cookies, cookies()} | {session_cookies, cookies()}
cookies() = [cookie()]
cookie() = term()
```

Produces a list of the entire cookie database. Intended for debugging/testing purposes. If no profile is specified, the default profile is used.

```
which_sessions() -> session_info()
which_sessions(Profile) -> session_info()
```

Types:

```
Profile = profile() | pid()
```

When started `stand_alone` only the pid can be used.

```
session_info() = {GoodSessions, BadSessions, NonSessions}
```

```
GoodSessions = session()
BadSessions = tuple()
NonSessions = term()
```

Produces a slightly processed dump of the session database. It is intended for debugging. If no profile is specified, the default profile is used.

SEE ALSO

RFC 2616, *inets(3)*, *gen_tcp(3)*, *ssl(3)*

httpd

Erlang module

An implementation of an HTTP 1.1 compliant web server, as defined in **RFC 2616**. Provides web server start options, administrative functions, and an Erlang callback API.

DATA TYPES

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

`boolean() = true | false`

`string()` = list of ASCII characters

`path() = string()` representing a file or a directory path

`ip_address() = {N1,N2,N3,N4} % IPv4 | {K1,K2,K3,K4,K5,K6,K7,K8} % IPv6`

`hostname() = string()` representing a host, for example, "foo.bar.com"

`property() = atom()`

ERLANG HTTP SERVER SERVICE START/STOP

A web server can be configured to start when starting the `Inets` application, or dynamically in runtime by calling the `Inets` application API `inets:start(httpd, ServiceConfig)` or `inets:start(httpd, ServiceConfig, How)`, see *inets(3)*. The configuration options, also called properties, are as follows:

File Properties

When the web server is started at application start time, the properties are to be fetched from a configuration file that can consist of a regular Erlang property list, that is, `[{Option, Value}]`, where `Option = property()` and `Value = term()`, followed by a full stop, or for backwards compatibility, an Apache-like configuration file. If the web server is started dynamically at runtime, a file can still be specified but also the complete property list.

{proplist_file, path()}

If this property is defined, `Inets` expects to find all other properties defined in this file. The file must include all properties listed under mandatory properties.

{file, path()}

If this property is defined, `Inets` expects to find all other properties defined in this file, which uses Apache-like syntax. The file must include all properties listed under mandatory properties. The Apache-like syntax is the property, written as one word where each new word begins with a capital, followed by a white-space, followed by the value, followed by a new line.

Example:

```
{server_root, "/urs/local/www"} -> ServerRoot /usr/local/www
```

A few exceptions are documented for each property that behaves differently, and the special cases `{directory, {path(), PropertyList}}` and `{security_directory, {Dir, PropertyList}}`, are represented as:

```
<Directory Dir>
 <Properties handled as described above>
</Directory>
```

Note:

The properties `proplist_file` and `file` are mutually exclusive. Also newer properties may not be supported as Apache-like options, this is a legacy feature.

Mandatory Properties

{port, integer()}

The port that the HTTP server listen to. If zero is specified as port, an arbitrary available port is picked and function `httpd:info/2` can be used to determine which port was picked.

{server_name, string()}

The name of your server, normally a fully qualified domain name.

{server_root, path()}

Defines the home directory of the server, where log files, and so on, can be stored. Relative paths specified in other properties refer to this directory.

{document_root, path()}

Defines the top directory for the documents that are available on the HTTP server.

Communication Properties

{bind_address, ip_address() | hostname() | any}

Default is `any`. `any` is denoted `*` in the Apache-like configuration file.

{profile, atom()}

Used together with *`bind_address`* and *`port`* to uniquely identify a HTTP server. This can be useful in a virtualized environment, where there can be more that one server that has the same bind_address and port. If this property is not explicitly set, it is assumed that the *`bind_address`* and *`port`*uniquely identifies the HTTP server.

{socket_type, ip_comm | {ip_comm, Config::proplist()} | {essl, Config::proplist()}}

For `ip_comm` configuration options, see *gen_tcp:listen/2*, some options that are used internally by httpd can not be set.

For `SSL` configuration options, see *ssl:listen/2*.

Default is `ip_comm`.

{ipfamily, inet | inet6}

Default is `inet`, legacy option `inet6fb4` no longer makes sense and will be translated to inet.

{minimum_bytes_per_second, integer()}

If given, sets a minimum of bytes per second value for connections.

If the value is unreached, the socket closes for that connection.

The option is good for reducing the risk of "slow DoS" attacks.

Erlang Web Server API Modules

{modules, [atom()]}

Defines which modules the HTTP server uses when handling requests. Default is `[mod_alias, mod_auth, mod_esi, mod_actions, mod_cgi, mod_dir, mod_get, mod_head, mod_log, mod_disk_log]`. Notice that some `mod`-modules are dependent on others, so the order cannot be entirely arbitrary. See the *Inets Web Server Modules* in the User's Guide for details.

Limit properties

{customize, atom()}

A callback module to customize the inets HTTP servers behaviour see *httpd_custom_api*

{disable_chunked_transfer_encoding_send, boolean()}

Allows you to disable chunked transfer-encoding when sending a response to an HTTP/1.1 client. Default is `false`.

{keep_alive, boolean()}

Instructs the server whether to use persistent connections when the client claims to be HTTP/1.1 compliant. Default is `true`.

{keep_alive_timeout, integer()}

The number of seconds the server waits for a subsequent request from the client before closing the connection. Default is `150`.

{max_body_size, integer()}

Limits the size of the message body of an HTTP request. Default is no limit.

{max_clients, integer()}

Limits the number of simultaneous requests that can be supported. Default is `150`.

{max_header_size, integer()}

Limits the size of the message header of an HTTP request. Default is `10240`.

{max_content_length, integer()}

Maximum content-length in an incoming request, in bytes. Requests with content larger than this are answered with status 413. Default is `100000000` (100 MB).

{max_uri_size, integer()}

Limits the size of the HTTP request URI. Default is no limit.

{max_keep_alive_request, integer()}

The number of requests that a client can do on one connection. When the server has responded to the number of requests defined by `max_keep_alive_requests`, the server closes the connection. The server closes it even if there are queued request. Default is no limit.

{max_client_body_chunk, integer()}

Enforces chunking of a HTTP PUT or POST body data to be deliverd to the mod_esi callback. Note this is not supported for mod_cgi. Default is no limit e.i the whole body is deliverd as one entity, which could be very memory consuming. *mod_esi(3)*.

Administrative Properties

{mime_types, [{MimeType, Extension}] | path()}

`MimeType = string()` and `Extension = string()`. 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 can be specified as an Apache-like file or directly in the property list. Such a file can look like the follwoing:

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

Default is [{"html","text/html"},{"htm","text/html"}].

{mime_type, string()}

When the server is asked to provide a document type that cannot be determined by the MIME Type Settings, the server uses this default type.

{server_admin, string()}

Defines the email-address of the server administrator to be included in any error messages returned by the server.

{server_tokens, none|prod|major|minor|minimal|os|full|{private, string()}}

Defines the look of the value of the server header.

Example: Assuming the version of `Inets` is 5.8.1, the server header string can look as follows for the different values of server-tokens:

`none`

"" % A Server: header will not be generated

`prod`

"inets"

`major`

"inets/5"

`minor`

"inets/5.8"

`minimal`

"inets/5.8.1"

`os`

"inets/5.8.1 (unix)"

`full`

"inets/5.8.1 (unix/linux) OTP/R15B"

`{private, "foo/bar"}`

"foo/bar"

By default, the value is as before, that is, `minimal`.

{log_format, common | combined}

Defines if access logs are to be written according to the `common` log format or the extended common log format. The `common` format is one line looking like this: `remotehost rfc931 authuser [date] "request" status bytes`.

Here:

`remotehost`
: Remote.

`rfc931`
: The remote username of the client (**RFC 931**).

`authuser`
: The username used for authentication.

`[date]`
: Date and time of the request (**RFC 1123**).

`"request"`
: The request line 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.

The `combined` format is one line looking like this: `remotehost rfc931 authuser [date] "request" status bytes "referer" "user_agent"`

In addition to the earlier:

`"referer"`
: The URL the client was on before requesting the URL (if it could not be determined, a minus sign is placed in this field).

`"user_agent"`
: The software the client claims to be using (if it could not be determined, a minus sign is placed in this field).

This affects the access logs written by `mod_log` and `mod_disk_log`.

{error_log_format, pretty | compact}

Default is `pretty`. If the error log is meant to be read directly by a human, `pretty` is the best option.

`pretty` has a format corresponding to:

```
io:format("[~s] ~s, reason: ~n ~p ~n~n", [Date, Msg, Reason]).
```

`compact` has a format corresponding to:

```
io:format("[~s] ~s, reason: ~w ~n", [Date, Msg, Reason]).
```

This affects the error logs written by `mod_log` and `mod_disk_log`.

URL Aliasing Properties - Requires mod_alias

{alias, {Alias, RealName}}

`Alias = string()` and `RealName = string()`. `alias` allows documents to be stored in the local file system instead of the `document_root` location. URLs with a path beginning with url-path is mapped to local files beginning with directory-filename, for example:

```
{alias, {"/image", "/ftp/pub/image"}}
```

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

{re_write, {Re, Replacement}}

`Re = string()` and `Replacement = string()`. `re_write` allows documents to be stored in the local file system instead of the `document_root` location. URLs are rewritten by `re:replace/3` to produce a path in the local file-system, for example:

```
{re_write, {"^/[~]([^/]+)(.*)$", "/home/\\1/public\\2"}}
```

Access to http://your.server.org/~bob/foo.gif would refer to the file /home/bob/public/foo.gif. In an Apache-like configuration file, `Re` is separated from `Replacement` with one single space, and as expected backslashes do not need to be backslash escaped, the same example would become:

```
ReWrite ^/[~]([^/]+)(.*)$ /home/\1/public\2
```

Beware of trailing space in `Replacement` to be used. If you must have a space in `Re`, use, for example, the character encoding `\040`, see *re(3)*.

{directory_index, [string()]}

`directory_index` 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 can be given, in which case the server returns the first it finds, for example:

```
{directory_index, ["index.html", "welcome.html"]}
```

Access to http://your.server.org/docs/ would return http://your.server.org/docs/index.html or http://your.server.org/docs/welcome.html if index.html does not exist.

CGI Properties - Requires mod_cgi

{script_alias, {Alias, RealName}}

`Alias = string()` and `RealName = string()`. Have the same behavior as property `alias`, except that they also mark 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:

```
{script_alias, {"/cgi-bin/", "/web/cgi-bin/"}}
```

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

{script_re_write, {Re, Replacement}}

`Re = string()` and `Replacement = string()`. Have the same behavior as property `re_write`, except that they also mark 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:

```
{script_re_write, {"^/cgi-bin/(\\d+)/", "/web/\\1/cgi-bin/"}}
```

Access to http://your.server.org/cgi-bin/17/foo would cause the server to run the script /web/17/cgi-bin/foo.

{script_nocache, boolean()}

If `script_nocache` is set to `true`, the HTTP server by default adds the header fields necessary to prevent proxies from caching the page. Generally this is preferred. Default to `false`.

{script_timeout, integer()}

The time in seconds the web server waits between each chunk of data from the script. If the CGI script does not deliver any data before the timeout, the connection to the client is closed. Default is `15`.

{action, {MimeType, CgiScript}} - requires mod_action

`MimeType = string()` and `CgiScript = string()`. `action` adds an action activating 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.

Example:

```
{action, {"text/plain", "/cgi-bin/log_and_deliver_text"}}
```

{script, {Method, CgiScript}} - requires mod_action

`Method = string()` and `CgiScript = string()`. `script` adds an action activating 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.

Example:

```
{script, {"PUT", "/cgi-bin/put"}}
```

ESI Properties - Requires mod_esi

{erl_script_alias, {URLPath, [AllowedModule]}}

`URLPath = string()` and `AllowedModule = atom()`. `erl_script_alias` marks all URLs matching url-path as erl scheme scripts. A matching URL is mapped into a specific module and function, for example:

```
{erl_script_alias, {"/cgi-bin/example", [httpd_example]}}
```

A request to http://your.server.org/cgi-bin/example/httpd_example:yahoo would refer to httpd_example:yahoo/3 or, if that does not exist, httpd_example:yahoo/2 and http://your.server.org/cgi-bin/example/other:yahoo would not be allowed to execute.

{erl_script_nocache, boolean()}

If `erl_script_nocache` is set to `true`, the server adds HTTP header fields preventing proxies from caching the page. This is generally a good idea for dynamic content, as the content often varies between each request. Default is `false`.

{erl_script_timeout, integer()}

If `erl_script_timeout` sets the time in seconds the server waits between each chunk of data to be delivered through `mod_esi:deliver/2`. Default is `15`. This is only relevant for scripts that use the erl scheme.

{eval_script_alias, {URLPath, [AllowedModule]}}

`URLPath = string()` and `AllowedModule = atom()`. Same as `erl_script_alias` but for scripts using the eval scheme. This is only supported for backwards compatibility. The eval scheme is deprecated.

Log Properties - Requires mod_log

{error_log, path()}

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 `server_root`.

{security_log, path()}

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 `server_root`.

{transfer_log, path()}

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 `server_root`.

Disk Log Properties - Requires mod_disk_log

{disk_log_format, internal | external}

Defines the file format of the log files. See `disk_log` for details. If the internal file format is used, the log file is repaired after a crash. When a log file is repaired, data can disappear. When the external file format is used, `httpd` does not start if the log file is broken. Default is `external`.

{error_disk_log, path()}

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 `server_root`.

{error_disk_log_size, {MaxBytes, MaxFiles}}

`MaxBytes = integer()` and `MaxFiles = integer()`. Defines the properties of the (`disk_log(3)`) error log file. This file is of type wrap log and max bytes is written to each file and max files is used before the first file is truncated and reused.

{security_disk_log, path()}

Defines the filename of the (`disk_log(3)`) access log file logging incoming security events, that is, authenticated requests. If the filename does not begin with a slash (/), it is assumed to be relative to the `server_root`.

{security_disk_log_size, {MaxBytes, MaxFiles}}

`MaxBytes = integer()` and `MaxFiles = integer()`. Defines the properties of the `disk_log(3)` access log file. This file is of type wrap log and max bytes is written to each file and max files is used before the first file is truncated and reused.

{transfer_disk_log, path()}

Defines the filename of the (`disk_log(3)`) access log file logging incoming requests. If the filename does not begin with a slash (/), it is assumed to be relative to the `server_root`.

{transfer_disk_log_size, {MaxBytes, MaxFiles}}

`MaxBytes = integer()` and `MaxFiles = integer()`. Defines the properties of the `disk_log(3)` access log file. This file is of type wrap log and max bytes is written to each file and max files is used before the first file is truncated and reused.

Authentication Properties - Requires mod_auth

{directory, {path(), [{property(), term()}]}}

The properties for directories are as follows:

{allow_from, all | [RegxpHostString]}

Defines a set of hosts to be granted access to a given directory, 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.

{deny_from, all | [RegxpHostString]}

Defines a set of hosts to be denied access to a given directory, 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 not allowed access.

{auth_type, plain | dets | mnesia}

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 are used. This property is called `AuthDbType` in the Apache-like configuration files.

{auth_user_file, path()}

Sets the name of a file containing the list of users and passwords for user authentication. The filename can be either absolute or relative to the `server_root`. If using the plain storage method, this file is a plain text file where each line contains a username followed by a colon, followed by the non-encrypted password. If usernames are duplicated, the behavior is undefined.

Example:

```
ragnar:s7Xxv7
edward:wwjau8
```

If the Dets storage method is used, the user database is maintained by Dets and must not be edited by hand. Use the API functions in module `mod_auth` to create/edit the user database. This directive is ignored if the Mnesia storage method is used. For security reasons, ensure that `auth_user_file` is stored outside the document tree of the web server. If it is placed in the directory that it protects, clients can download it.

{auth_group_file, path()}

Sets the name of a file containing the list of user groups for user authentication. The filename can be either absolute or relative to the `server_root`. If the plain storage method is used, the group file is a plain text file, where each line contains a group name followed by a colon, followed by the members usernames separated by spaces.

Example:

```
group1: bob joe ante
```

If the Dets storage method is used, the group database is maintained by Dets and must not be edited by hand. Use the API for module `mod_auth` to create/edit the group database. This directive is ignored if the Mnesia storage method is used. For security reasons, ensure that the `auth_group_file` is stored outside the document tree of the web server. If it is placed in the directory that it protects, clients can download it.

{auth_name, string()}

Sets the name of the authorization realm (auth-domain) for a directory. This string informs the client about which username and password to use.

{auth_access_password, string()}

If 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 web server is started. Otherwise it is written in clear text in the configuration file.

{require_user, [string()]}

Defines users to grant access to a given directory using a secret password.

{require_group, [string()]}

Defines users to grant access to a given directory using a secret password.

Htaccess Authentication Properties - Requires mod_htaccess

{access_files, [path()]}

Specifies the filenames that are used for access files. When a request comes, every directory in the path to the requested asset are searched after files with the names specified by this parameter. If such a file is found, the file is parsed and the restrictions specified in it are applied to the request.

Security Properties - Requires mod_security

{security_directory, {path(), [{property(), term()}]}}

The properties for the security directories are as follows:

{data_file, path()}

Name of the security data file. The filename can either be absolute or relative to the `server_root`. This file is used to store persistent data for module `mod_security`.

{max_retries, integer()}

Specifies the maximum number of attempts to authenticate a user before the user is blocked out. If a user successfully authenticates while blocked, the user receives a 403 (Forbidden) response from the server. If the user makes a failed attempt while blocked, the server returns 401 (Unauthorized), for security reasons. Default is `3`. Can be set to infinity.

{block_time, integer()}

Specifies the number of minutes a user is blocked. After this timehas passed, the user automatically regains access. Default is `60`.

{fail_expire_time, integer()}

Specifies the number of minutes a failed user authentication is remembered. If a user authenticates after this time has passed, the previous failed authentications are forgotten. Default is `30`.

{auth_timeout, integer()}

Specifies the number of seconds a successful user authentication is remembered. After this time has passed, the authentication is no longer reported. Default is `30`.

Exports

```
info(Pid) ->
info(Pid, Properties) -> [{Option, Value}]
```

Types:

```
Properties = [property()]
Option = property()
Value = term()
```

Fetches information about the HTTP server. When called with only the pid, all properties are fetched. When called with a list of specific properties, they are fetched. The available properties are the same as the start options of the server.

Note:

Pid is the pid returned from `inets:start/[2,3]`. Can also be retrieved form `inets:services/0` and `inets:services_info/0`, see *inets(3)*.

```
info(Address, Port) ->
info(Address, Port, Profile) ->
info(Address, Port, Profile, Properties) -> [{Option, Value}]
info(Address, Port, Properties) -> [{Option, Value}]
```

Types:

```
Address = ip_address()
Port = integer()
Profile = atom()
Properties = [property()]
Option = property()
Value = term()
```

Fetches information about the HTTP server. When called with only `Address` and `Port`, all properties are fetched. When called with a list of specific properties, they are fetched. The available properties are the same as the start options of the server.

Note:

The address must be the IP address and cannot be the hostname.

```
reload_config(Config, Mode) -> ok | {error, Reason}
```

Types:

```
Config = path() | [{Option, Value}]
Option = property()
Value = term()
Mode = non_disturbing | disturbing
```

Reloads the HTTP server configuration without restarting the server. Incoming requests are answered with a temporary down message during the reload time.

Note:

Available properties are the same as the start options of the server, but the properties `bind_address` and `port` cannot be changed.

If mode is disturbing, the server is blocked forcefully, all ongoing requests terminates, and the reload starts immediately. If mode is non-disturbing, no new connections are accepted, but ongoing requests are allowed to complete before the reload is done.

ERLANG WEB SERVER API DATA TYPES

The Erlang web server API data types are as follows:

```
    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
}).
```

To acess the record in your callback-module use:

```
-include_lib("inets/include/httpd.hrl").
```

The fields of record `mod` have 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 an IP socket or an `ssl` socket.

`socket`

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

`config_db`

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

`method`

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

`absolute_uri`

If the request is an HTTP/1.1 request, the URI can be in the absolute URI format. In that case, `httpd` saves the absolute URI in this field. An Example of an absolute URI is `"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 is 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 can be in lower case or upper case. `httpd` ensures that all header field names are in lower 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 connection and is not closed when the request is served.

ERLANG WEB SERVER API CALLBACK FUNCTIONS

Exports

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

Types:

```
OldData = list()
NewData = [{response,{StatusCode,Body}}]
| [{response,{response,Head,Body}}]
| [{response,{already_sent,Statuscode,Size}}]
StatusCode = integer()
Body = io_list() | nobody | {Fun, Arg}
Head = [HeaderOption]
HeaderOption = {Option, Value} | {code, StatusCode}
Option = accept_ranges | allow
| cache_control | content_MD5
| content_encoding | content_language
| content_length | content_location
| content_range | content_type | date
| etag | expires | last_modified
| location | pragma | retry_after
| server | trailer | transfer_encoding
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 configuration option of `Module`. The function can 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 is the list returned from the last call to `do/1`.

`Body` is the body of the HTTP response that is sent back to the client. An appropriate header is appended to the message. `StatusCode` is the status code of the response, see **RFC 2616** for the appropriate values.

`Head` is a key value list of HTTP header fields. The server constructs an HTTP header from this data. See **RFC 2616** for the appropriate value for each header field. If the client is an HTTP/1.0 client, the server filters the list so that only HTTP/1.0 header fields are sent back to the client.

If `Body` is returned and equal to `{Fun,Arg}`, the web server tries `apply/2` on `Fun` with `Arg` as argument. The web server expects that the fun either returns a list `(Body)` that is an 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 signals this to the client by closing the connection.

`Module:load(Line, AccIn)-> eof | ok | {ok, AccOut} | {ok, AccOut, {Option, Value}} | {ok, AccOut, [{Option, Value}]} | {error, Reason}`

Types:

```
Line = string()
AccIn = [{Option, Value}]
AccOut = [{Option, Value}]
Option = property()
Value = term()
Reason = term()
```

Converts a line in an Apache-like configuration file to an `{Option, Value}` tuple. Some more complex configuration options, such as `directory` and `security_directory`, create an accumulator. This function only needs clauses for the options implemented by this particular callback module.

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

Types:

```
ConfigDB = ets_table()
Reason = term()
```

When `httpd` is shut down, it tries to execute `remove/1` in each Erlang web server callback module. The programmer can use this function to clean up resources created in the store function.

`Module:store({Option, Value}, Config)-> {ok, {Option, NewValue}} | {error, Reason}`

Types:

```
Line = string()
Option = property()
Config = [{Option, Value}]
Value = term()
Reason = term()
```

Checks the validity of the configuration options before saving them in the internal database. This function can also have a side effect, that is, setup of necessary extra resources implied by the configuration option. It can also resolve possible dependencies among configuration options by changing the value of the option. This function only needs clauses for the options implemented by this particular callback module.

ERLANG WEB SERVER API HELP FUNCTIONS

Exports

`parse_query(QueryString) -> [{Key,Value}]`

Types:

```
QueryString = string()
```

```
Key = string()
Value = string()
```

`parse_query/1` parses incoming data to `erl` and `eval` scripts (see *mod_esi(3)*) as defined in the standard URL format, that is, '+' becomes 'space' and decoding of hexadecimal characters (`%xx`).

SEE ALSO

RFC 2616, *inets(3)*, *ssl(3)*

httpd_custom_api

Erlang module

The module implementing this behaviour shall be supplied to to the servers configuration with the option *customize*

Exports

`response_default_headers() -> [Header]`

Types:

`Header = {HeaderName :: string(), HeaderValue::string()}`

string:to_lower/1 will be performed on the HeaderName

Provide default headers for the HTTP servers responses. Note that this option may override built-in defaults.

`response_header({HeaderName, HeaderValue}) -> {true, Header} | false`

Types:

`Header = {HeaderName :: string(), HeaderValue::string()}`

The header name will be in lower case and should not be altered.

Filter and possible alter HTTP response headers before they are sent to the client.

`request_header({HeaderName, HeaderValue}) -> {true, Header} | false`

Types:

`Header = {HeaderName :: string(), HeaderValue::string()}`

The header name will be in lower case and should not be altered.

Filter and possible alter HTTP request headers before they are processed by the server.

httpd_socket

Erlang module

This module provides the Erlang web server 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 `Data` over `Socket` using the specified `SocketType`. `Socket` and `SocketType` is to 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/2` 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)

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 defined in **RFC 2616**.

create_etag(FileInfo) -> Etag

Types:

FileInfo = file_info()
Etag = string()

`create_etag/1` calculates the Etag for a file from its size and time for last modification. `FileInfo` is a record defined in `kernel/include/file.hrl`.

day(NthDayOfWeek) -> DayOfWeek

Types:

NthDayOfWeek = 1-7
DayOfWeek = string()

`day/1` converts the day of the week (`NthDayOfWeek`) from an integer (1-7) to an abbreviated string, that is:

1 = "Mon", 2 = "Tue", ..., 7 = "Sat".

decode_hex(HexValue) -> DecValue

Types:

HexValue = DecValue = string()

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

flatlength(NestedList) -> Size

Types:

NestedList = list()
Size = integer()

`flatlength/1` computes the size of the possibly nested list `NestedList`, which can contain binaries.

hexlist_to_integer(HexString) -> Number

Types:

Number = integer()
HexString = string()

`hexlist_to_integer` converts 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 representing `Number` in a hexadecimal form.

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 file `mime.types` (located in the config directory).

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

Types:

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

`lookup_mime_default` returns the MIME type associated with a specific file suffix as specified in the `mime.types` file (located in the config directory). If no appropriate association is 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 `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()
```

`reason_phrase` returns `Description` of an HTTP 1.1 `StatusCode`, for example, 200 is "OK" and 201 is "Created". For more information, see **RFC 2616**.

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 `String` in `N` chunks using `RegExp`. `split/3` is equivalent to `regexp:split/2` with the exception that `N` defines the maximum number of fields in `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, then `not_a_script` is returned.

split_path(RequestLine) -> {Path,QueryStringOrPathInfo}

Types:

RequestLine = Path = QueryStringOrPathInfo = string()

`split_path/1` splits `RequestLine` in a file reference (`Path`), and a `QueryString` or a `PathInfo` string as specified in **RFC 2616**. A `QueryString` is isolated from `Path` with a question mark (?) and `PathInfo` with a slash (/). In the case of a `QueryString`, everything before ? is a `Path` and everything after ? is a `QueryString`. In the case of a `PathInfo`, `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 is to be read as horizontal tab or space.

suffix(FileName) -> Suffix

Types:

FileName = Suffix = string()

`suffix/1` is equivalent to `filename:extension/1` with the exception that `Suffix` is returned without a leading dot (`.`).

SEE ALSO

httpd(3)

mod_alias

Erlang module

Erlang web server internal API for handling of, for example, interaction data exported by module `mod_alias`.

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 User's Guide*.

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

Types:

`PathData = interaction_data()`
`ConfigDB = config_db()`
`RequestURI = Path = string()`

`path/3` returns the 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 User's Guide*.

`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, `ShortPath` and `AfterPath`, as defined in *httpd_util:split_path/1*. `Path` is generated from `ShortPath`, that is, the result from *default_index/2* with `ShortPath` as an argument. `config_db()` is the server config file in ETS table format as described in *Inets User's Guide*.

`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_script_name/3` traverses `ScriptAliases`, typically extracted from `ConfigDB`, and matches each `FakeName` with `RequestURI`. If a match is found, `FakeName` is replaced with `RealName` in the match. If the resulting match is not an executable script, `not_a_script` is returned. If it is a script, the resulting script path is in two parts, `ShortPath` and `AfterPath`, as defined in *httpd_util:split_script_path/1*. `config_db()` is the server config file in ETS table format as described in *Inets User's Guide*.

mod_auth

Erlang module

This module provides for basic user authentication using textual files, Dets databases, or Mnesia databases.

Exports

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` each 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, options `Port` and `Dir` are mandatory.

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` each adds a user to the user database. If the operation is successful, this function returns `true`. If an error occurs, `{error,Reason}` is returned. When `add_user/2` is called, options `Password`, `UserData`, `Port`, and `Dir` 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` each deletes the group specified and returns `true`. If there is an error, `{error, Reason}` is returned. When `delete_group/2` is called, option `Port` and `Dir` 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` each deletes a user from a group. If the group or the user does not exist, this function returns an error, otherwise `true`. When `delete_group_member/3` is called, the options `Port` and `Dir` are mandatory.

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

Types:

```
UserName = string()
```

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

`delete_user/2`, `delete_user/3`, and `delete_user/4` each deletes a user from the user database. If the operation is successful, this function returns `true`. If an error occurs, `{error,Reason}` is returned. When `delete_user/2` is called, options `Port` and `Dir` 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` each returns an `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, options `Port` and `Dir` 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` each lists all the groups available. If there is an error, `{error, Reason}` is returned. When `list_groups/1` is called, options `Port` and `Dir` 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` each 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, options `Port` and `Dir` are mandatory.

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

Types:

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

`list_users/1`, `list_users/2`, and `list_users/3` each returns a list of users in the user database for a specific `Port/Dir`. When `list_users/1` is called, options `Port` and `Dir` 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` each updates `AuthAccessPassword` for the specified directory. If `NewPassword` is equal to "NoPassword", no password is required to change authorisation data. If `NewPassword` is equal to "DummyPassword", no changes can be done without changing the password first.

SEE ALSO

httpd(3), mod_alias(3)

mod_esi

Erlang module

This module defines the Erlang Server Interface (ESI) API. It is a more efficient way of writing Erlang scripts for your `Inets` web server than writing them as common CGI scripts.

DATA TYPES

The following data types are used in the functions for mod_esi:

`env() =`

`{EnvKey()::atom(), Value::term()}`

Currently supported key value pairs

`{server_software, string()}`

Indicates the inets version.

`{server_name, string()}`

The local hostname.

`{gateway_interface, string()}`

Legacy string used in CGI, just ignore.

`{server_protocol, string()}`

HTTP version, currently "HTTP/1.1"

`{server_port, integer()}`

Servers port number.

`{request_method, "GET | "PUT" | "DELETE" | "POST" | "PATCH"}`

HTTP request method.

`{remote_adress, inet:ip_address()}`

The clients ip address.

`{peer_cert, undefined | no_peercert | DER:binary()}`

For TLS connections where client certificates are used this will be an ASN.1 DER-encoded X509-certificate as an Erlang binary. If client certificates are not used the value will be `no_peercert`, and if TLS is not used (HTTP or connection is lost due to network failure) the value will be `undefined`.

`{script_name, string()}`

Request URI

`{http_LowerCaseHTTPHeaderName, string()}`

example: {http_content_type, "text/html"}

Exports

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

Types:

`SessionID = term()`

```
Data = string() | io_list() | binary()
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 an Erl Scheme script back to the client.

Note:

If any HTTP header fields are added by the script, they must be in the first call to `deliver/2`, and the data in the call must be a string. Calls after the headers are complete can contain binary data to reduce copying overhead. Do not assume anything about the data type of `SessionID`. `SessionID` must be the value given as input to the ESI callback function that you implemented.

ESI Callback Functions

Exports

```
Module:Function(SessionID, Env, Input)-> {continue, State} | _
```

Types:

```
SessionID = term()
Env = env()
Input = string() | chunked_data()
chunked_data() = {first, Data::binary()} | {continue, Data::binary(),
State::term()} | {last, Data::binary(), State::term()}
State = term()
```

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

`mod_esi:deliver/2` shall be used to generate the response to the client and `SessionID` is an identifier that shall by used when calling this function, do not assume anything about the datatype. This function may be called several times to chunk the response data. Notice 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 does not contain the **end of HTTP header**, that is, `"\r\n\r\n"`, the server assumes that no HTTP header fields will be generated.

`Env` environment data of the request see description above.

`Input` is query data of a GET request or the body of a PUT or POST request. The default behavior (legacy reasons) for delivering the body, is that the whole body is gathered and converted to a string. But if the httpd config parameter *max_client_body_chunk* is set, the body will be delivered as binary chunks instead. The maximum size of the chunks is either *max_client_body_chunk* or decide by the client if it uses HTTP chunked encoding to send the body. When using the chunking mechanism this callback must return {continue, State::term()} for all calls where `Input` is `{first, Data::binary()}` or `{continue, Data::binary(), State::term()}`. When `Input` is `{last, Data::binary(), State::term()}` the return value will be ignored.

Note:

Note that if the body is small all data may be delivered in only one chunk and then the callback will be called with {last, Data::binary(), undefined} without getting called with `{first, Data::binary()}`.

The input `State` is the last returned `State`, in it the callback can include any data that it needs to keep track of when handling the chunks.

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

Types:

```
Env = env()
Input = string()
Response = string()
```

This callback format consumes much memory, as the whole response must be generated before it is sent to the user. This callback format is deprecated. For new development, use `Module:Function/3`.

mod_security

Erlang module

Security Audit and Trailing Functionality

Exports

```
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` each blocks the user `User` from directory `Dir` for a specified amount of time.

```
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` each returns a list of users that are currently authenticated. Authentications are stored for `SecurityAuthTimeout` seconds, and 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` each returns a list of users that are currently blocked from access.

```
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` each removes the user `User` from the list of blocked users for `Port` (and `Dir`).

SecurityCallbackModule

The `SecurityCallbackModule` is a user-written module that can receive events from the `mod_security` Erlang web server API module. This module only exports the functions event/[4,5] which are described here.

Exports

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

Types:

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

`event/4` or `event/5` is called whenever an event occurs in the `mod_security` Erlang web server API module. (`event/4` is called if `Address` is undefined, otherwise `event/5`. Argument `What` specifies the type of event that has occurred and is one of the following reasons:

`auth_fail`

A failed user authentication.

`user_block`

A user is being blocked from access.

`user_unblock`

A user is being removed from the block list.

Note:

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

http_uri

Erlang module

This module provides utility functions for working with URIs, according to **RFC 3986**.

DATA TYPES

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

`boolean() = true | false`

`string()` = list of ASCII characters

URI DATA TYPES

Type definitions that are related to URI:

`uri() = string() | binary()`

Syntax according to the URI definition in RFC 3986, for example, "http://www.erlang.org/"

`user_info() = string() | binary()`

`scheme() = atom()`

Example: http, https

`host() = string() | binary()`

`port() = inet:port_number()`

`path() = string() | binary()`

Represents a file path or directory path

`query() = string() | binary()`

`fragment() = string() | binary()`

For more information about URI, see **RFC 3986**.

Exports

`decode(HexEncodedURI) -> URI`

Types:

`HexEncodedURI = string() | binary() - A possibly hexadecimal encoded URI`
`URI = uri()`

Decodes a possibly hexadecimal encoded URI.

`encode(URI) -> HexEncodedURI`

Types:

`URI = uri()`
`HexEncodedURI = string() | binary() - Hexadecimal encoded URI`

Encodes a hexadecimal encoded URI.

```
parse(URI) -> {ok, Result} | {error, Reason}
parse(URI, Options) -> {ok, Result} | {error, Reason}
```

Types:

```
URI = uri()
Options = [Option]
Option = {ipv6_host_with_brackets, boolean()} | {scheme_defaults,
scheme_defaults()} | {fragment, boolean()} | {scheme_validation_fun,
fun()}
Result = {Scheme, UserInfo, Host, Port, Path, Query} | {Scheme, UserInfo,
Host, Port, Path, Query, Fragment}
Scheme = scheme()
UserInfo = user_info()
Host = host()
Port = inet:port_number()
Path = path()
Query = query()
Fragment = fragment()
Reason = term()
```

Parses a URI. If no scheme defaults are provided, the value of the *scheme_defaults* function is used.

When parsing a URI with an unknown scheme (that is, a scheme not found in the scheme defaults), a port number must be provided, otherwise the parsing fails.

If the fragment option is `true`, the URI fragment is returned as part of the parsing result, otherwise it is ignored.

Scheme validation fun is to be defined as follows:

```
fun(SchemeStr :: string() | binary()) ->
 valid | {error, Reason :: term()}.
```

It is called before scheme string gets converted into scheme atom and thus possible atom leak could be prevented

Warning:

The scheme portion of the URI gets converted into atom, meaning that atom leak may occur. Specifying a scheme validation fun is recommended unless the URI is already sanitized.

```
scheme_defaults() -> SchemeDefaults
```

Types:

```
SchemeDefaults = [{scheme(), default_scheme_port_number()}]
default_scheme_port_number() = inet:port_number()
```

Provides a list of the scheme and their default port numbers supported (by default) by this utility.