inets

inets 5.3
August 2 2010

August 2 2010

1 User's Guide

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

1.1 Introduction

1.1.1 Purpose

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

1.1.2 Prerequisites

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

1.1.3 The Service Concept

Each client and server in inets is viewed as service. Services may be configured to be started at application startup or started dynamically in runtime. If you want to run inets as an distributed application that should handle application failover and takeover, services should be configured to be started at application startup. When starting the inets application the inets top supervisor will start a number of subsupervisors and worker processes for handling the different services provided. When starting services dynamically new children will be added to the supervision tree, unless the service is started with the stand alone option, in which case the service is linked to the calling process and all OTP application features such as soft upgrade are lost.

Services that should be configured for startup at application startup time should be put into the erlang node configuration file on the form:

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

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

1.2 FTP Client

1.2.1 Introduction

Ftp clients are consider to be rather temporary and are for that reason only started and stopped during runtime and can not be started at application startup. Due to the design of FTP client API, letting some functions return intermediate results, only the process that started the ftp client will be able to access it in order to preserve sane semantics. (This could be solved by changing the API and using the concept of a controlling process more in line with other OTP applications, but that is perhaps something for the future.) If the process that started the ftp session dies the ftp client process will terminate.

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

1.2.2 Using the FTP Client API

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

```
1> 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
```

1.3 HTTP Client

1.3.1 Introduction

The HTTP client default profile will be started when the inets application is started and is then available to all processes on that erlang node. Other profiles may also be started at application startup, or profiles can be started and stopped dynamically in runtime. Each client profile will spawn a new process to handle each request unless there is a possibility to use a persistent connection with or without pipelining. The client will add a host header and an empty te header if there are no such headers present in the request.

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

1.3.2 Configuration

What to put in the erlang node application configuration file in order to start a profile at application startup.

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

For valid properties see *http(3)*

1.3.3 Using the HTTP Client API

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

The following calls uses the default client profile. Use the proxy "www-proxy.mycompany.com:8000", but not for requests to localhost. This will apply to all subsequent requests

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

An ordinary synchronous request.

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

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

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

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

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

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

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

Send a request with a specified connection header.

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

Start a HTTP client profile.

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

The new profile has no proxy settings so the connection will be refused

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

Stop a HTTP client profile.

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

Alternatively:

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

1.4 HTTP server

1.4.1 Introduction

The HTTP server, also referred to as httpd, handles HTTP requests as described in RFC 2616 with a few exceptions such as gateway and proxy functionality. The server supports ipv6 as long as the underlying mechanisms also do so.

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

The configuration of the server is provided as an erlang property list, and for backwards compatibility also a configuration file using apache-style configuration directives is supported.

As of inets version 5.0 the HTTP server is an easy to start/stop and customize web server that provides the most basic web server functionality. Depending on your needs there are also other erlang based web servers that may be of interest such as Yaws, http://yaws.hyber.org, that for instance has its own markup support to generate html, and supports certain buzzword technologies such as SOAP.

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

1.4.2 Configuration

What to put in the erlang node application configuration file in order to start a 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 also apache-like config files are supported.

All possible config 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()]
```

{proplist_file, file()} File containing an erlang property list, followed by a full stop, describing the HTTP server configuration.

{file, file()} If you use an old apace-like configuration file.

{debug, debug()} - Can enable trace on all functions or only exported functions on chosen modules.

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

1.4.3 Using the HTTP Server API

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

Start a HTTP server with minimal required configuration. Note that if you specify port 0 an arbitrary available port will be used and you can use the info function to find out 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}
```

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

```
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.
```

```
  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}).
```

Note that bind_address has to be the ip address reported by the info function and can not be the hostname that is allowed when inputting bind_address.

1.4.4 Htaccess - User Configurable Authentication.

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

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

Access Files Directives

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

- *DIRECTIVE: "allow"*

  Syntax:`Allow` from subnet subnet|from all

Default:`from all`

Same as the directive allow for the server config file.

- *DIRECTIVE: "AllowOverRide"*

  Syntax:`AllowOverRide` all | none | Directives
  Default:`- None -`
  `AllowOverRide` Specify which parameters that not access-files in subdirectories are allowed to alter the value for. If the parameter is set to none no more access-files will be parsed.

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

- *DIRECTIVE: "AuthGroupfile"*

  Syntax:`AuthGroupFile` Filename
  Default:`- None -`

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

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

- *DIRECTIVE: "AuthName"*

  Syntax:`AuthName` auth-domain
  Default:`- None -`

  Same as the directive AuthName for the server config file.

- *DIRECTIVE: "AuthType"*

  Syntax:`AuthType` Basic
  Default:`Basic`

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

- *DIRECTIVE: "AuthUserFile"*

  Syntax:`AuthUserFile` Filename
  Default:`- None -`

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

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

- *DIRECTIVE: "deny"*

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

  Same as the directive deny for the server config file.

- *DIRECTIVE: "Limit"*

  Syntax:`<Limit` RequestMethods`>`
  Default: - None -

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

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

- *DIRECTIVE: "order"*
  Syntax:`order` allow deny | deny allow
  Default: allow deny

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

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

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

  Syntax:`require` group group1 group2...|user user1 user2...
  Default:`- None -`
  Context: Limit

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

1.4.5 Dynamic Web Pages

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

First there are CGI-scripts that can be written in any programming language. CGI-scripts are standardized and supported by most 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.

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

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

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

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

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

```
plain text"
```

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

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

Erlang Server Interface (ESI)

The erlang server interface is implemented by the module mod_esi.

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

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

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

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

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

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

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

The module (Mod) referred to must be found in the code path, and data returned by the function (Func) is passed back to the client. Data returned from the function must furthermore take the form as specified in the CGI specification. See *mod_esi(3)* for implementation details of the esi callback function.

Note:

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

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

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

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

1.4.6 Logging

There are three types of logs supported. Transfer logs, security logs and error logs. The de-facto standard Common Logfile Format is used for the transfer and security logging. There are numerous statistics programs available to analyze Common Logfile Format. The Common Logfile Format looks as follows:

remotehost rfc931 authuser [date] "request" status bytes

remotehost
: Remote hostname

rfc931
: The client's remote username (RFC 931).

authuser
: The username with which the user authenticated himself.

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

"request"
: The request line exactly as it came from the client (RFC 1945).

status
: The HTTP status code returned to the client (RFC 1945).

bytes
: The content-length of the document transferred.

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

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

1.4.7 Server Side Includes

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

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

SERVER-SIDE INCLUDES (SSI) SETUP

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

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

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

```
text/x-server-parsed-html html htm
```

Server-Side Includes (SSI) Format

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

```
<!--#command tag1="value1" tag2="value2" -->
```

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

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

`errmsg`

controls the message sent back to the client if an error occurred while parsing the document. All errors are logged in the server's error log.

`sizefmt`

determines the format used to display the size of a file. Valid choices are `bytes` or `abbrev`. `bytes` for a formatted byte count or `abbrev` for an abbreviated version displaying the number of kilobytes.

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

`virtual`

gives a virtual path to a document on the server. Only normal files and other parsed documents can be accessed in this way.

`file`

gives a pathname relative to the current directory. `../` cannot be used in this pathname, nor can absolute paths. As above, you can send other parsed documents, but you cannot send CGI scripts.

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

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

The lastmod directive prints the last modification date of the specified file. Valid tags are the same as with the `include` command.

The exec directive executes a given shell command or CGI script. Valid tags are:

`cmd`

executes the given string using `/bin/sh`. All of the variables defined below are defined, and can be used in the command.

`cgi`

executes the given virtual path to a CGI script and includes its output. The server does not perform error checking on the script output.

Server-Side Includes (SSI) Environment Variables

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

`DOCUMENT_NAME`

The current filename.

`DOCUMENT_URI`

The virtual path to this document (such as `/docs/tutorials/foo.shtml`).

`QUERY_STRING_UNESCAPED`

The unescaped version of any search query the client sent, with all shell-special characters escaped with `\`.

`DATE_LOCAL`

The current date, local time zone.

`DATE_GMT`

Same as DATE_LOCAL but in Greenwich mean time.

`LAST_MODIFIED`

The last modification date of the current document.

1.4.8 The Erlang Web Server API

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

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

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

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

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

API Description

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

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

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

1.4.9 Inets Web Server Modules

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

mod_action - Filetype/Method-Based Script Execution.

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

Uses the following Erlang 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

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

Exports the following Erlang 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

This 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 user name with which the user has authenticated himself.

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

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

first_start() ->
    mnesia:create_schema([node()]),
```

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

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

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

mod_cgi - CGI Scripts

This module handles invoking of CGI scripts

mod_dir - Directories

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

Uses the following Erlang 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

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

Uses the following Erlang 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 user name with which the user has authenticated himself.

mod_include - SSI

This module makes it possible to expand "macros" embedded in HTML pages before they are delivered to the client, that is Server-Side Includes (SSI).

Uses the following Erlang Webserver API interaction data:

- real_name - from mod_alias
- remote_user - from mod_auth

Exports the following Erlang Webserver API interaction data:

`{mime_type, MimeType}`
The file suffix of the incoming URL mapped into a `MimeType` as defined in the Mime Type Settings section.

mod_log - Logging Using Text Files.

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

Uses the following Erlang Webserver API interaction data:

- remote_user - from mod_auth

mod_range - Requests with Range Headers

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

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

Uses the following Erlang Webserver API interaction data:

- real_name - from mod_alias

mod_response_control - Requests with If* Headers

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

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

Uses the following Erlang Webserver API interaction data:

- real_name - from mod_alias

Exports the following Erlang Webserver API interaction data:

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

mod_security - Security Filter

This module serves as a filter for authenticated requests handled in mod_auth. It provides possibility to restrict users from access for a specified amount of time if they fail to authenticate several times. It logs failed authentication as well as blocking of users, and it also calls a configurable call-back module when the events occur.

There is also an API to manually block, unblock and list blocked users or users, who have been authenticated within a configurable amount of time.

mod_trace - TRACE Request

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

2 Reference Manual

Inets is a container for Internet clients and servers. Currently a FTP client, a HTTP client and server, and a tftp client and server has been 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.

COMMON DATA TYPES

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

```
service() = ftpc | tfptd | 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` will not be 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 a [{Option, Value}] list. The information given in the list is specific for each service and it is probable that each service will have its own info function that gives you even 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)*

`stop() -> ok`

Stops the inets application. 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 will not be handled by application takeover and failover behavior when inets is run as a distributed application. Nor will they be automatically restarted when the inets application is restarted, but as long as the inets application is up and running they will be supervised and may be soft code upgraded. Services started as `stand_alone`, e.i. the service is not started as part of the inets application, will lose all OTP application benefits such as soft upgrade. The "stand_alone-service" will be linked to the process that started it. In most cases some of the supervision functionality will still be in place and in some sense the calling process has now become the top supervisor.

`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 the `stand_alone` option is used in start, only the pid is a valid argument to stop.

SEE ALSO

ftp(3), http(3), httpd(3), tftp(3)

ftp

Erlang module

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

Starting from inets version 4.4.1 the ftp client will always try to use passive ftp mode and only resort to active ftp mode if this fails. There is a start option *mode* where this default behavior may be changed.

There are two ways to start an ftp client. One is using the *Inets service framework* and the other is to start it directy as a standalone process using the *open* function.

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

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

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

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)`. See *inets(3)* for more info.

Below follows a description of the available configuration options.

{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()`

This determines if the FTP communication should 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 (it tries to use IPv6 and only if that does not work, it uses IPv4).

Default is `inet` (IPv4).

{timeout, Timeout}

Timeout = `integer() >= 0`

Connection timeout.

Default is 60000 (milliseconds).

{progress, Progress}

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

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

InitProgress = `term()`

Default is `ignore`.

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

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

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

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

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

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

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

The callback function should be defined as

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

CBModule = CBFunction = atom()

UserProgressTerm = term()

File = string()

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

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

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

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

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

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

COMMON DATA TYPES

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

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

string() = list of ASCII characters.

shortage_reason() = etnospc | epnospc

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

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

Exports

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

Types:

Pid = pid()

Account = string()

Reason = eacct | common_reason()

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

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

Types:

Pid = pid()

LocalFile = RemoteFile = string()

Reason = epath | elogin | etnospc | epnospc | efnamena | common_reason

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

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

Types:

Pid = pid()

Bin = binary()()

RemoteFile = string()

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

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

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

Types:

Pid = pid()

Bin = binary()

Reason = echunk | restriction_reason() | common_reason()

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

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

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

Types:

Pid = pid()

File = string()

Reason = restriction_reason() | common_reason()

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

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

Types:

Pid = pid()

Reason = echunk | restriction_reason() | shortage_reason()

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

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

Types:

Pid = pid()

Dir = string()

Reason = restriction_reason() | common_reason()

Changes the working directory at the remote server to `Dir`.

`close(Pid) -> ok`

Types:

Pid = pid()

Ends an ftp session, created using the *open* function.

`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 even a file. The `Pathname` string can contain wildcard(s).

`ls/1` implies the user's current remote directory.

The format of `Listing` is operating system dependent (on UNIX it is typically produced from the output of the `ls -l` shell command).

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

Types:

Pid = pid()

Dir = string()

Reason = restriction_reason() | common_reason()

Creates the directory `Dir` at the remote server.

```
nlist(Pid) ->
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 even a file. The `Pathname` string can contain wildcard(s).

`nlist/1` implies the user's current remote directory.

The format of `Listing` is a stream of file names, where each name is separated by <CRLF> or <NL>. Contrary to the `ls` function, the purpose of `nlist` is to make it possible for a program to automatically process file name information.

```
open(Host) -> {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() (defaults to false)

debug() = disable | debug | trace (defaults to disable)

open_option() = {ipfamily, ipfamily()} | {port, port()} | {mode, mode()} | {timeout, timeout()} | {progress, progress()}

ipfamily() = inet | inet6 | inet6fb4 (defaults to inet)

port() = integer() > 0 (defaults to 21)

mode() = active | passive (defaults to passive)

timeout() = integer() >= 0 (defaults to 60000 milliseconds)

pogress() = ignore | {module(), function(), initial_data()} (defaults to ignore)

module() = atom()

function() = atom()

initial_data() = term()

Reason = ehost | term()

This function is used to start a standalone ftp client process (without the inets service framework) and open a session with the FTP server at `Host`.

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

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

Types:

Pid = pid()

Reason = restriction_reason() | common_reason()

Returns the current working directory at the remote server.

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

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

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

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

Types:

Pid = pid()

Bin = binary()

RemoteFile = string()

Reason = restriction_reason() | common_reason()

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

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

Types:

Pid = pid()

RemoteFile = string()

Reason = restriction_reason() | common_reason()

Start transfer of the file `RemoteFile` from the remote server.

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

Types:

Pid = pid()

Bin = binary()

Reason = restriction_reason() | common_reason()

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

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

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

Types:

Pid = pid()

CurrFile = NewFile = string()

Reason = restriction_reason() | common_reason()

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

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

Types:

Pid = pid()

Dir = string()

Reason = restriction_reason() | common_reason()

Removes directory `Dir` at the remote server.

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

Types:

Pid = pid()

LocalFile = RemoteFile = string()

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

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

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

Types:

Pid = pid()

Bin = binary()()

RemoteFile = string()

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

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

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

Types:

Pid = pid()

Bin = binary()

Reason = echunk | restriction_reason() | common_reason()

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

Note that for some errors, e.g. file system full, it is 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()

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

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

Types:

Pid = pid()

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

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

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

Types:

Pid = pid()

Type = ascii | binary

Reason = etype | restriction_reason() | common_reason()

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

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

Types:

Pid = pid()

User = Password = string()

Reason = euser | common_reason()

Performs login of `User` with `Password`.

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

Types:

Pid = pid()

User = Password = string()

Reason = euser | common_reason()

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

quote(Pid, Command) -> [FTPLine]

Types:

Pid = pid()

Command = string()

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

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

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

ERRORS

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

`echunk`

Synchronisation error during chunk sending.

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

`eclosed`

The session has been closed.

`econn`

Connection to remote server prematurely closed.

`ehost`

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

`elogin`

User not logged in.

`enotbinary`

Term is not a binary.

`epath`

No such file or directory, or directory already exists, or permission denied.

`etype`

No such type.

`euser`

User name or password not valid.

`etnospc`

Insufficient storage space in system [452].

`epnospc`

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

`efnamena`

File name not allowed [553].

SEE ALSO

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

tftp

Erlang module

This is a complete implementation of the following IETF standards:

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

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

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

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

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

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 already is 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 below in the *COMMON 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.

COMMON DATA TYPES

```
ServiceConfig = Options
Options = [option()]
option() -- see below
```

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

`{debug, Level}`

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

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

`{host, Host}`

`Host = hostname()` see *inet(3)*

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

`{port, Port}`

`Port = int()`

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

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

`{port_policy, Policy}`

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

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

`{udp, Options}`

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

`{use_tsize, Bool}`

`Bool = bool()`

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

`{max_tsize, MaxTsize}`

`MaxTsize = int() | infinity`

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

`{max_conn, MaxConn}`

`MaxConn = int() | infinity`

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

`{TftpKey, TftpVal}`

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

The name and value of a TFTP option.

`{reject, Feature}`

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

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

`{callback, {RegExp, Module, State}}`

```
RegExp = string()
Module = atom()
State = term()
```

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

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

`{logger, Module}`

```
Module = module()()
```

Callback module for customized logging of error, warning and info messages. >The callback module must implement the `tftp_logger` behavior, *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 will try to resend a message up to `5` times when the timeout expires.

Exports

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

Types:

Options = [option()]

Pid = pid()

Reason = term()

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

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

Types:

RemoteFilename = string()

LocalFilename = binary | string()

Options = [option()]

LastCallbackState = term()

Reason = term()

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

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

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

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

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

Types:

RemoteFilename = string()

LocalFilename = binary() | string()

Options = [option()]

LastCallbackState = term()

Reason = term()

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

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

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

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

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

Types:

Pid = [pid()()]

Options = [option()]

Reason = term()

Returns info about all TFTP daemon processes.

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

Types:

Pid = [pid()()]

Options = [option()]

Reason = term()

Returns info about all TFTP server processes.

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

Types:

Options = [option()]

Reason = term()

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

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

Types:

Options = [option()]

Pid = pid()

Result = ok | {error, Reason}

Reason = term()

Changes config for all TFTP daemon processes

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

Types:

Options = [option()]

Pid = pid()

Result = ok | {error, Reason}

Reason = term()

Changes config for all TFTP server processes

`change_config(Pid, Options) -> Result`

Types:

Pid = pid()

Options = [option()]

Result = ok | {error, Reason}

Reason = term()

Changes config for a TFTP daemon, server or client process

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

Types:

Reason = term()

Starts the Inets application.

CALLBACK FUNCTIONS

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

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

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

If a callback (which performs the file access in the TFTP server) takes too long time (more than the double TFTP timeout), the server will abort the connection and send an error reply to the client. This implies that the server will release 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 will simply ignore the new request if the request is equal with 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 will consumer lesser resources.

Exports

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

Types:

Peer = {PeerType, PeerHost, PeerPort}

PeerType = inet | inet6

PeerHost = ip_address()

PeerPort = integer()

Access = read | write

Filename = string()

Mode = string()

SuggestedOptions = AcceptedOptions = [{Key, Value}]

Key = Value = string()

InitialState = [] | [{root_dir, string()}]

NewState = term()

Code = undef | enoent | eacces | enospc
| badop | eexist | baduser | badopt
| int()

Text = string()

Prepares to open a file on the client side.

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

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

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

Types:

Peer = {PeerType, PeerHost, PeerPort}

PeerType = inet | inet6

PeerHost = ip_address()

PeerPort = integer()

Access = read | write

Filename = string()

Mode = string()

SuggestedOptions = AcceptedOptions = [{Key, Value}]
Key = Value = string()
State = InitialState | term()
InitialState = [] | [{root_dir, string()}]
NewState = term()
Code = undef | enoent | eacces | enospc
| badop | eexist | baduser | badopt
| int()
Text = string()

Opens a file for read or write access.

On the client side where the `open/5` call has been 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 may be added, but the ones that are present in `SuggestedOptions` may be omitted or replaced with new values in `AcceptedOptions`.

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

Types:

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

Read a chunk from the file.

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

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

Types:

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

Write a chunk to the file.

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

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

Types:

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

Invoked when the file transfer is aborted.

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

LOGGER FUNCTIONS

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

Exports

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

Types:

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

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

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

Types:

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

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

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

Types:

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

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

httpc

Erlang module

This module provides the API to a HTTP/1.1 compatible client according to RFC 2616, caching is currently not supported.

> ### Note:
>
> When starting the Inets application a manager process for the default profile will be started. The functions in this API that does not explicitly use a profile will 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 needs to be started.
>
> Also note that pipelining will only be used if the pipeline timeout is set, otherwise persistent connections without pipelining will be used e.i. the client always waits for the previous response before sending the next request.

There are some usage examples in the *Inets User's Guide*.

COMMON DATA TYPES

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

```
boolean()    = true | false
string()     = list of ASCII characters
request_id() = ref()
profile()    = atom()
path()       = string() representing a file path or directory path
ip_address() = See inet(3)
socket_opt() = See the Options used by gen_tcp(3) and
               ssl(3) connect(s)
```

HTTP DATA TYPES

Type definitions that are related to HTTP:

For more information about HTTP see rfc 2616

```
method()        = head | get | put | post | trace | options | delete
request()       = {url(), headers()} |
                  {url(), headers(), content_type(), body()}
url()           = string() - Syntax according to the URI definition in rfc 2396, ex: "http://www.erlang.org"
status_line()   = {http_version(), status_code(), reason_phrase()}
http_version()  = string() ex: "HTTP/1.1"
status_code()   = integer()
reason_phrase() = string()
content_type()  = string()
headers()       = [header()]
header()        = {field(), value()}
field()         = string()
value()         = string()
```

```
body()          = string() | binary()
filename()      = string()
```

SSL DATA TYPES

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

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

HTTP CLIENT SERVICE START/STOP

A 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)* Below follows a description of the available configuration options.

{profile, profile()}
: Name of the profile, see common data types below, this option is mandatory.

{data_dir, path()}
: Directory where the profile may save persistent data, if omitted all cookies will be treated as session cookies.

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

Exports

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

Types:

Url = url()

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

Profile = profile()

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, ssl_options()} | {autoredirect, boolean()} | {proxy_auth, {userstring(), passwordstring()}} | {version, http_version()} | {relaxed, 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()}}

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()

Profile = profile()

Reason = term()

Sends a HTTP-request. The function can be both synchronous and asynchronous. In the later case the function will return {ok, RequestId} and later on the information will be delivered to the `receiver` depending on that value.

Http option (`http_option()`) details:

`timeout`

Timeout time for the request.

The clock start ticking as soon as the request has been sent.

Time is in milliseconds.

Defaults to `infinity`.

`connect_timeout`

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

Time is in milliseconds.

Defaults to the value of the `timeout` option.

`ssl`

If using SSL, these SSL-specific options are used.

Defaults to `[]`.

`autoredirect`

Should the client automatically retreive the information from the new URI and return that as the result instead of a 30X-result code.

Note that for some 30X-result codes automatic redirect is not allowed in these cases the 30X-result will always be returned.

Defaults to `true`.

`proxy_auth`

A proxy-authorization header using the provided user name and password will be 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 will not be used.

Defaults to the trsing `"HTTP/1.1"`.

`relaxed`

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

Defaults to `false`.

Option (`option()`) details:

`sync`

Shall the request be synchronous or asynchronous.

Defaults to `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 the option `self` the the following stream messages will be sent to that process: {http, {RequestId, stream_start, Headers}, {http, {RequestId, stream, BinBodyPart}, {http, {RequestId, stream_end, Headers}. When streaming to to the calling processes using the option `{self, once}` the first message will have an additional element e.i. {http, {RequestId, stream_start, Headers, Pid}, this is the process id that should be used as an argument to http:stream_next/1 to trigger the next message to be sent to the calling process.

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

Defaults to `none`.

`body_format`

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

Defaults to `string`.

`full_result`

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

Defaults to `true`.

`header_as_is`

Shall the headers provided by the user be made lower case or be regarded as case sensitive.

Note that the http standard requires them to be case insenstive. This feature should only be used if there is no other way to communicate with the server or for testing purpose. Also note that when this option is used no headers will be automatically added, all necessary headers has to be provided by the user.

Defaults to `false`.

`socket_opts`

Socket options to be used for this and subsequent request(s).

Overrides any value set by the *set_options* function.

Note that the validity of the options are *not* checked in any way.

Note that this may change the socket behaviour (see *inet:setopts/2*) for an already existing, and therefor already connected request handler.

By defaults the socket options set by the *set_options/1,2* function is used when establishing connection.

`receiver`

Defines how the client will deliver the result for a asynchroneous request (`sync` has the value `false`).

`pid()`

Message(s) will be sent to this process in the format:

```
{http, ReplyInfo}
```

`function/1`

Information will be delivered to the receiver via calls to the provided fun:

```
Receiver(ReplyInfo)
```

`{Module, Funcion, Args}`

Information will be delivered to the receiver via calls to the callback function:

```
apply(Module, Function, [ReplyInfo | Args])
```

In all cases above, `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}
```

Defaults to the `pid()` of the process calling the request function (`self()`).

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

Types:

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

Profile = profile()

Cancels an asynchronous HTTP-request.

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

Types:

Options = [Option]

Option = {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}

Proxy = {Hostname, Port}

Hostname = string()

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

Port = integer()

ex: 8080

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!

NoProxy = [NoProxyDesc]

NoProxyDesc = DomainDesc | HostName | IPDesc

DomainDesc = "*.Domain"

ex: "*.ericsson.se"

IpDesc = string()

ex: "134.138" or "[FEDC:BA98" (all IP-addresses starting with 134.138 or FEDC:BA98), "66.35.250.150" or "[2010:836B:4179::836B:4179]" (a complete IP-address).

MaxSessions = integer()

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

MaxKeepAlive = integer()

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

KeepAliveTimeout = integer()

Default is *120000* (= 2 min). If a persistent connection is idle longer than the keep_alive_timeout the client will close the connection. The server may also have a such a time out but you should not count on it!

MaxPipeline = integer()

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

PipelineTimeout = integer()

Default is *0*, which will result in pipelining not being used. If a persistent connection is idle longer than the pipeline_timeout the client will close the connection.

CookieMode = enabled | disabled | verify

Default is *disabled*. If Cookies are enabled all valid cookies will automatically be saved in the client manager's cookie database. If the option verify is used the function http:verify_cookie/2 has to be called for the cookie to be saved.

IpFamily = inet | inet6 | inet6fb4

By default *inet*. When it is set to `inet6fb4` you can use both ipv4 and ipv6. It first tries `inet6` and if that does not works falls back to `inet`. The option is here to provide a workaround for buggy ipv6 stacks to ensure that ipv4 will always work.

IpAddress = ip_address()

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

Port = integer()

Specify which local port number to use. See gen_tcp:connect/3,4 for more info.

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()

Sets options to be used for subsequent requests.

> Note:
>
> If possible the client will keep its connections alive and use 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 would be ideal to be sent on a persistent connection, this very much depends on the application. Note that a very long queue of requests may cause a user perceived delays as earlier request may take a long time to complete. The HTTP/1.1 specification does suggest a limit of 2 persistent connections per server, which is the default value of the max_sessions option.

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

Types:

Pid = pid() - as received in the stream_start message

Triggers the next message to be streamed, e.i. same behavior as active once for sockets.

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

Types:

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

Url = url()

Profile = profile()

Saves the cookies defined in SetCookieHeaders in the client profile's cookie database. You need to call this function if you set the option cookies to `verify`. If no profile is specified the default profile will be used.

```
cookie_header(Url) ->
cookie_header(Url, Profile) -> header() | {error, Rason}
```

Types:

Url = url()

Profile = profile()

Returns the cookie header that would be sent when making a request to Url using the profile Profile. If no profile is specified the default profile will be used.

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

Types:

Profile = profile()

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

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

`which_cookies(Profile) -> cookies()`

Types:

Profile = profile()

cookies() = [cooie_stores()]

cookie_stores() = {cookies, icookies()} | {session_cookies, icookies()}

icookies() = [icookie()]

cookie() = term()

This function produces a list of the entire cookie database. It is intended for debugging/testing purposes. If no profile is specified the default profile will be used.

SEE ALSO

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

httpd

Erlang module

Documents the HTTP server start options, some administrative functions and also specifies the Erlang Web server callback API

COMMON 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 directory path.

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

hostname() = string() - representing a host ex "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 started dynamically in runtime by calling the Inets application API `inets:start(httpd, ServiceConfig)`, or `inets:start(httpd, ServiceConfig, How)`, see *inets(3)* Below follows a description of the available configuration options, also called properties.

File properties

When the web server is started at application start time the properties should be fetched from a configuration file that could consist of a regular erlang property list, e.i. `[{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 you may still specify a file but you could also just specify the complete property list.

{proplist_file, path()}
: If this property is defined inets will expect to find all other properties defined in this file. Note that the file must include all properties listed under mandatory properties.

{file, path()}
: If this property is defined inets will expect to find all other properties defined in this file, that uses Apache like syntax. Note that 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. Ex:

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

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

```
<Directory Dir>
```

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

> **Note:**
>
> The properties proplist_file and file are mutually exclusive.

Mandatory properties

{port, integer()}
: The port that the HTTP server shall listen on. If zero is specified as port, an arbitrary available port will be picked and you can use the httpd:info/2 function to find out which port was picked.

{server_name, string()}
: The name of your server, normally a fully qualified domain name.

{server_root, path()}
: Defines the servers home directory where log files etc 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}
: Defaults to `any`. Note that `any` is denoted * in the apache like configuration file.

{socket_type, ip_comm | ssl}

: Defaults to `ip_comm`.

{ipfamily, inet | inet6 | inet6fb4}

: Defaults to `inet6fb4.`

: Note that this option is only used when the option `socket_type` has the value `ip_comm`.

Erlang Web server API modules

{modules, [atom()]}
: Defines which modules the HTTP server will use to handle requests. Defaults to: `[mod_alias, mod_auth, mod_esi, mod_actions, mod_cgi, mod_dir, mod_get, mod_head, mod_log, mod_disk_log]` Note that some mod-modules are dependent on others, so the order can not be entirely arbitrary. See the *Inets Web server Modules in the Users guide* for more information.

Limit properties

{disable_chunked_transfer_encoding_send, boolean()}
: This property allows you to disable chunked transfer-encoding when sending a response to a HTTP/1.1 client, by default this is false.

{keep_alive, boolean()}
: Instructs the server whether or not 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 will wait 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 HTTP request. By the default there is no limit.

{max_clients, integer()}
Limits the number of simultaneous requests that can be supported. Defaults to 150.
{max_header_size, integer()}
Limits the size of the message header of HTTP request. Defaults to 10240.
{max_uri, integer()}
Limits the size of the HTTP request URI. By default there is no limit.
{max_keep_alive_requests, integer()}
The number of request 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 close the connection. The server will close it even if there are queued request. Defaults to no limit.

Administrative properties

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

Where 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 as well as directly in the property list. Such a file may look like:

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

Defaults to [{"html","text/html"},{"htm","text/html"}]

{mime_type, string()}
When the server is asked to provide a document type which cannot be determined by the MIME Type Settings, the server will use this default type.
{server_admin, string()}
ServerAdmin defines the email-address of the server administrator, to be included in any error messages returned by the server.
{log_format, common | combined}

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

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

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

```
"referer"
 The url the client was on before
 requesting your url. (If it could not be determined a minus
 sign will be placed in this field)
"user_agent"
 The software the client claims to be using. (If it
 could not be determined a minus sign will be placed in
 this field)
```

This affects the access logs written by mod_log and mod_disk_log.

{error_log_format, pretty | compact}

Defaults to pretty. If the error log is meant to be read directly by a human `pretty` will be the best option. `pretty` has the format corresponding to:

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

`compact` has the 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.

ssl properties

{ssl_ca_certificate_file, path()}
Used as cacertfile option in ssl:listen/2 see *ssl(3)*

{ssl_certificate_file, path()}
Used as certfile option in ssl:listen/2 see *ssl(3)*

{ssl_ciphers, list()}
Used as ciphers option in ssl:listen/2 see *ssl(3)*

{ssl_verify_client, integer()}
Used as verify option in ssl:listen/2 see *ssl(3)*

{ssl_verify_depth, integer()}
Used as depth option in ssl:listen/2 see *ssl(3)*

{ssl_password_callback_function, atom()}
Used together with ssl_password_callback_module to retrieve a value to use as password option to ssl:listen/2 see *ssl(3)*

{ssl_password_callback_arguments, list()}
Used together with ssl_password_callback_function to supply a list of arguments to the callback function. If not specified the callback function will be assumed to have arity 0.

{ssl_password_callback_module, atom()}
Used together with ssl_password_callback_function to retrieve a value to use as password option to ssl:listen/2 see *ssl(3)*

URL aliasing properties - requires mod_alias

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

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

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

{directory_index, [string()]}

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

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

and 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 do not exist.

CGI properties - requires mod_cgi

{script_alias, {Alias, RealName}}

Where Alias = string() and RealName = string(). Has the same behavior as the Alias property, except that it also marks the target directory as containing CGI scripts. URLs with a path beginning with url-path are mapped to scripts beginning with directory-filename, for example:

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

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

{script_nocache, boolean()}

If ScriptNoCache is set to true the HTTP server will by default add the header fields necessary to prevent proxies from caching the page. Generally this is something you want. Defaults to false.

{script_timeout, integer()}

The time in seconds the web server will wait between each chunk of data from the script. If the CGI-script not delivers any data before the timeout the connection to the client will be closed. Defaults to 15.

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

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

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

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

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

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

ESI properties - requires mod_esi

{erl_script_alias, {URLPath, [AllowedModule]}}

Where 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]}
```

and a request to http://your.server.org/cgi-bin/example/httpd_example:yahoo would refer to 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 will add http header fields that prevents proxies from caching the page. This is generally a good idea for dynamic content, since the content often vary between each request. Defaults to false.

{erl_script_timeout, integer()}
: If erl_script_timeout sets the time in seconds the server will wait between each chunk of data to be delivered through mod_esi:deliver/2. Defaults to 15. This is only relevant for scripts that uses the erl scheme.

{eval_script_alias, {URLPath, [AllowedModule]}}
: Where URLPath = string() and AllowedModule = atom(). Same as erl_script_alias but for scripts using the eval scheme. Note that 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 more information. If the internal file-format is used, the logfile will be repaired after a crash. When a log file is repaired data might get lost. When the external file-format is used httpd will not start if the log file is broken. Defaults to external.

{error_disk_log, internal | external}
: 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}}
: Where MaxBytes = integer() and MaxFiles = integer(). Defines the properties of the (disk_log(3)) error log file. The disk_log(3) error log file is of type wrap log and max-bytes will be written to each file and max-files will be used before the first file is truncated and reused.

{security_disk_log, path()}
: Defines the filename of the (disk_log(3)) access log file which logs incoming security events i.e authenticated requests. If the filename does not begin with a slash (/) it is assumed to be relative to the server_root.

{security_disk_log_size, {MaxBytes, MaxFiles}}
: Where MaxBytes = integer() and MaxFiles = integer(). Defines the properties of the disk_log(3) access log file. The disk_log(3) access log file is of type wrap log and max-bytes will be written to each file and max-files will be used before the first file is truncated and reused.

{transfer_disk_log, path()}
: Defines the filename of the (disk_log(3)) access log file which logs incoming requests. If the filename does not begin with a slash (/) it is assumed to be relative to the server_root.

{transfer_disk_log_size, {MaxBytes, MaxFiles}}
Where MaxBytes = integer() and MaxFiles = integer(). Defines the properties of the disk_log(3) access log file. The disk_log(3) access log file is of type wrap log and max-bytes will be written to each file and max-files will be used before the first file is truncated and reused.

Authentication properties - requires mod_auth

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

Here follows the valid properties for directories

{allow_from, all | [RegxpHostString]}
Defines a set of hosts which should 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 which should 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 is used. This property is called AuthDbType in the Apache like configuration files.
{auth_user_file, path()}
Sets the name of a file which contains the list of users and passwords for user authentication. filename can be either absolute or relative to the `server_root`. If using the plain storage method, this file is a plain text file, where each line contains a user name followed by a colon, followed by the non-encrypted password. If user names are duplicated, the behavior is undefined. For example:

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

If using the dets storage method, the user database is maintained by dets and should not be edited by hand. Use the API functions in mod_auth module to create / edit the user database. This directive is ignored if using the mnesia storage method. For security reasons, make sure that the `auth_user_file` is stored outside the document tree of the Web server. If it is placed in the directory which it protects, clients will be able to download it.
{auth_group_file, path()}
Sets the name of a file which contains the list of user groups for user authentication. Filename can be either absolute or relative to the `server_root`. If you use the plain storage method, the group file is a plain text file, where each line contains a group name followed by a colon, followed by the member user names separated by spaces. For example:

```
group1: bob joe ante
```

If using the dets storage method, the group database is maintained by dets and should not be edited by hand. Use the API for mod_auth module to create / edit the group database. This directive is ignored if using the mnesia storage method. For security reasons, make sure that the `auth_group_file` is stored outside

the document tree of the Web server. If it is placed in the directory which it protects, clients will be able to download it.

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

{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 since it otherwise is written in clear text in the configuration file.

{require_user, [string()]}
Defines users which should be granted access to a given directory using a secret password.

{require_group, [string()]}
Defines users which should be granted access to a given directory using a secret password.

Htaccess authentication properties - requires mod_htaccess

{access_files, [path()]}
Specify which filenames that are used for access-files. When a request comes every directory in the path to the requested asset will be searched after files with the names specified by this parameter. If such a file is found the file will be parsed and the restrictions specified in it will be applied to the request.

Security properties - requires mod_security

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

Here follows the valid properties for security directories

{security_data_file, path()}
Name of the security data file. The filename can either absolute or relative to the server_root. This file is used to store persistent data for the mod_security module.

{security_max_retries, integer()}
Specifies the maximum number of tries to authenticate a user has before the user is blocked out. If a user successfully authenticates when the user has been blocked, the user will receive a 403 (Forbidden) response from the server. If the user makes a failed attempt while blocked the server will return 401 (Unauthorized), for security reasons. Defaults to 3 may also be set to infinity.

{security_block_time, integer()}
Specifies the number of minutes a user is blocked. After this amount of time, he automatically regains access. Defaults to 60

{security_fail_expire_time, integer()}
Specifies the number of minutes a failed user authentication is remembered. If a user authenticates after this amount of time, his previous failed authentications are forgotten. Defaults to 30

{security_auth_timeout, integer()}
Specifies the number of seconds a successful user authentication is remembered. After this time has passed, the authentication will no longer be reported. Defaults to 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. Available properties are the same as the servers start options.

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

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

Types:

Address = ip_address()

Port = integer()

Properties = [property()]

Option = property()

Value = term()

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

> Note:
>
> Address has to be the ip-address and can not 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 will be answered with a temporary down message during the time the it takes to reload.

> Note:
>
> Available properties are the same as the servers start options, although the properties bind_address and port can not be changed.

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

ERLANG WEB SERVER API DATA TYPES

```
    ModData = #mod{}

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

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

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

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

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

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

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

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

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

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

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

`parsed_header`
: Type `[{HeaderKey,HeaderValue}]`, `parsed_header` contains all HTTP header fields from the HTTP-request stored in a list as key-value tuples. See RFC 2616 for a listing of all header fields. For example the date field would be stored as: `{"date","Wed, 15 Oct 1997 14:35:17 GMT"}. RFC 2616 defines that HTTP is a case insensitive protocol and the header fields may be in lower case or upper case. Httpd will ensure 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 will not be 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}]

StausCode = 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 Modules configuration option. The function may generate data for other modules or a response that can be sent back to the client.

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

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

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

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

`Module:load(Line, 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()

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

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

Types:

Line = string()

Option = property()

Config = [{Option, Value}]

Value = term()

Reason = term()

This function is used to check the validity of the configuration options before saving them in the internal database. This function may also have a side effect e.i. setup 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 does only need 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 shutdown it will try to execute `remove/1` in each Erlang web server callback module. The programmer may use this function to clean up resources that may have been created in the store function.

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_conf

Erlang module

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

Exports

`check_enum(EnumString,ValidEnumStrings) -> Result`

Types:

EnumString = string()

ValidEnumStrings = [string()]

Result = {ok,atom()} | {error,not_valid}

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

`clean(String) -> Stripped`

Types:

String = Stripped = string()

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

`custom_clean(String,Before,After) -> Stripped`

Types:

Before = After = regexp()

String = Stripped = string()

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

`is_directory(FilePath) -> Result`

Types:

FilePath = string()

Result = {ok,Directory} | {error,Reason}

Directory = string()

Reason = string() | enoent | eaccess | enotdir | FileInfo

FileInfo = File info record

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

`is_file(FilePath) -> Result`

Types:

FilePath = string()

Result = {ok,File} | {error,Reason}

File = string()

Reason = string() | enoent | eaccess | enotdir | FileInfo

FileInfo = File info record

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

`make_integer(String) -> Result`

Types:

String = string()

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

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

SEE ALSO

httpd(3)

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

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

Types:

SocketType = socket_type()

Socket = socket()

Port = integer()

IPAddress = string()

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

`resolve() -> HostName`

Types:

HostName = string()

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

SEE ALSO

httpd(3)

httpd_util

Erlang module

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

Exports

convert_request_date(DateString) -> ErlDate|bad_date

Types:

DateString = string()

ErlDate = {{Year,Month,Date},{Hour,Min,Sec}}

Year = Month = Date = Hour = Min = Sec = integer()

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

create_etag(FileInfo) -> Etag

Types:

FileInfo = file_info()

Etag = string()

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

decode_hex(HexValue) -> DecValue

Types:

HexValue = DecValue = string()

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

day(NthDayOfWeek) -> DayOfWeek

Types:

NthDayOfWeek = 1-7

DayOfWeek = string()

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

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

flatlength(NestedList) -> Size

Types:

NestedList = list()

Size = integer()

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

hexlist_to_integer(HexString) -> Number

Types:

Number = integer()

HexString = string()

`hexlist_to_integer` Convert the Hexadecimal value of HexString to an integer.

`integer_to_hexlist(Number) -> HexString`

Types:

Number = integer()

HexString = string()

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

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

Types:

ETSTable = ets_table()

Key = term()

Result = term() | undefined | Undefined

Undefined = term()

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

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

Types:

ConfigDB = ets_table()

Suffix = string()

MimeType = string() | undefined | Undefined

Undefined = term()

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

`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 can be found the value of DefaultType is returned.

`message(StatusCode,PhraseArgs,ConfigDB) -> Message`

Types:

StatusCode = 301 | 400 | 403 | 404 | 500 | 501 | 504

PhraseArgs = term()

ConfigDB = ets_table

Message = string()

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

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

`400 | 401 | 500`
`none` (No `PhraseArgs`)

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

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

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

`month(NthMonth) -> Month`

Types:

NthMonth = 1-12

Month = string()

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

1 = "Jan", 2 = "Feb", ..., 12 = "Dec".

`multi_lookup(ETSTable,Key) -> Result`

Types:

ETSTable = ets_table()

Key = term()

Result = [term()]

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

`reason_phrase(StatusCode) -> Description`

Types:

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

Description = string()

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

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

Types:

YYYY = MM = DD = Hour = Min =Sec = integer()

RFC1123Date = string()

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

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

Types:

String = RegExp = string()

SplitRes = {ok, FieldList} | {error, errordesc()}

Fieldlist = [string()]

N = integer

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

`split_script_path(RequestLine) -> Splitted`

Types:

RequestLine = string()

Splitted = not_a_script | {Path, PathInfo, QueryString}

Path = QueryString = PathInfo = string()

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

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

Types:

RequestLine = Path = QueryStringOrPathInfo = string()

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

`strip(String) -> Stripped`

Types:

String = Stripped = string()

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

`suffix(FileName) -> Suffix`

Types:

FileName = Suffix = string()

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

SEE ALSO

httpd(3)

mod_alias

Erlang module

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

Exports

`default_index(ConfigDB, Path) -> NewPath`

Types:

ConfigDB = config_db()

Path = NewPath = string()

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

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

Types:

PathData = interaction_data()

ConfigDB = config_db()

RequestURI = Path = string()

`path/3` returns the actual file `Path` in the `RequestURI` (See RFC 1945). If the interaction data `{real_name, {Path,AfterPath}}` has been exported by mod_alias; `Path` is returned. If no interaction data has been exported, ServerRoot is used to generate a file `Path`. `config_db()` and `interaction_data()` are as defined in *Inets Users Guide*.

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

Types:

ConfigDB = config_db()

RequestURI = string()

Aliases = [{FakeName,RealName}]

Ret = {ShortPath,Path,AfterPath}

ShortPath = Path = AfterPath = string()

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

mod_auth

Erlang module

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

Exports

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

Types:

UserName = string()

Options = [Option]

Option = {password,Password} | {userData,UserData} | {port,Port} | {addr,Address} | {dir,Directory} | {authPassword,AuthPassword}

Password = string()

UserData = term()

Port = integer()

Address = {A,B,C,D} | string() | undefined

Dir = string()

AuthPassword =string()

Reason = term()

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

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

Types:

UserName = string()

Options = [Option]

Option = {port,Port} | {addr,Address} | {dir,Directory} | {authPassword,AuthPassword}

Port = integer()

Address = {A,B,C,D} | string() | undefined

Dir = string()

AuthPassword = string()

Reason = term()

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

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

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

Types:

UserName = string()

Options = [Option]

Option = {port,Port} | {addr,Address} | {dir,Directory} | {authPassword,AuthPassword}

Port = integer()

Address = {A,B,C,D} | string() | undefined

Dir = string()

AuthPassword = string()

Reason = term()

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

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

Types:

Options = [Option]

Option = {port,Port} | {addr,Address} | {dir,Directory} | {authPassword,AuthPassword}

Port = integer()

Address = {A,B,C,D} | string() | undefined

Dir = string()

Users = list()

AuthPassword = string()

Reason = atom()

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

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

Types:

GroupName = string()

UserName = string()

Options = [Option]

Option = {port,Port} | {addr,Address} | {dir,Directory} | {authPassword,AuthPassword}

Port = integer()

Address = {A,B,C,D} | string() | undefined

Dir = string()

AuthPassword = string()

Reason = term()

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

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

Types:

GroupName = string()

UserName = string()

Options = [Option]

Option = {port,Port} | {addr,Address} | {dir,Directory} | {authPassword,AuthPassword}

Port = integer()

Address = {A,B,C,D} | string() | undefined

Dir = string()

AuthPassword = string()

Reason = term()

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

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

Types:

GroupName = string()

Options = [Option]

Option = {port,Port} | {addr,Address} | {dir,Directory} | {authPassword,AuthPassword}

Port = integer()

Address = {A,B,C,D} | string() | undefined

Dir = string()

Users = list()

AuthPassword = string()

Reason = term()

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

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

Types:

Options = [Option]

Option = {port,Port} | {addr,Address} | {dir,Directory} | {authPassword,AuthPassword}

Port = integer()

Address = {A,B,C,D} | string() | undefined

Dir = string()

Groups = list()

AuthPassword = string()

Reason = term()

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

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

Types:

Options = [Option]

Option = {port,Port} | {addr,Address} | {dir,Directory} | {authPassword,AuthPassword}

Port = integer()

Address = {A,B,C,D} | string() | undefined

Dir = string()

GroupName = string()

AuthPassword = string()

Reason = term()

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

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

Types:

Port = integer()

Address = {A,B,C,D} | string() | undefined

Dir = string()

GroupName = string()

OldPassword = string()

NewPassword = string()

Reason = term()

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

SEE ALSO

httpd(3), mod_alias(3),

mod_esi

Erlang module

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

Exports

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

Types:

SessionID = term()

Data = string() | io_list()

Reason = term()

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

Sends data from a Erl Scheme script back to the client.

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

ESI Callback Functions

Exports

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

Types:

SessionID = term()

Env = [EnvironmentDirectives] ++ ParsedHeader

EnvironmentDirectives = {Key,Value}

Key = query_string | content_length | server_software | gateway_interface | server_protocol | server_port | request_method | remote_addr | script_name. <v>Input = string()

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

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

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

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

Use this callback function to dynamically generate dynamic web content. when a part of the page is generated send the data back to the client through `deliver/2`. Note that the first chunk of data sent to the client must at least contain

all HTTP header fields that the response will generate. If the first chunk not contains *End of HTTP header* that is `"\r\n\r\n"` the server will assume that no HTTP header fields will be generated.

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

Types:

Env = [EnvironmentDirectives] ++ ParsedHeader

EnvironmentDirectives = {Key,Value}

Key = query_string | content_length | server_software | gateway_interface | server_protocol | server_port | request_method | remote_addr | script_name. <v>Input = string()

Response = string()

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

mod_security

Erlang module

Security Audit and Trailing Functionality

Exports

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

Types:

Port = integer()

Address = {A,B,C,D} | string() | undefined

Dir = string()

Users = list() = [string()]

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

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

Types:

Port = integer()

Address = {A,B,C,D} | string() | undefined

Dir = string()

Users = list() = [string()]

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

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

Types:

User = string()

Port = integer()

Address = {A,B,C,D} | string() | undefined

Dir = string()

Seconds = integer() | infinity

Reason = no_such_directory

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

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

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

Types:

User = string()

Port = integer()

Address = {A,B,C,D} | string() | undefined

Dir = string()

Reason = term()

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

The SecurityCallbackModule

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

Exports

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

Types:

What = atom()

Port = integer()

Address = {A,B,C,D} | string() <v>Dir = string()

What = [Info]

Info = {Name, Value}

`event/4` or `event/4` is called whenever an event occurs in the mod_security Erlang Webserver API module (`event/4` is called if Address is undefined and `event/5` otherwise). The `What` argument specifies the type of event that has occurred, and should be one of the following reasons; `auth_fail` (a failed user authentication), `user_block` (a user is being blocked from access) or `user_unblock` (a user is being removed from the block list).

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