Inets

version 4.7
Typeset in \LaTeX \ from SGML source using the DOCBUILDER 3.3.2 Document System.
Contents

1 Inets User’s Guide ................................. 1
  1.1 Introduction .......................................................... 1
    1.1.1 Purpose ............................................................ 1
    1.1.2 Prerequisites ..................................................... 1
    1.1.3 The Service Concept ............................................. 1
  1.2 FTP Client ....................................................... 2
    1.2.1 Introduction ..................................................... 2
    1.2.2 Using the FTP Client API ...................................... 2
  1.3 HTTP Client ....................................................... 2
    1.3.1 Introduction ..................................................... 2
    1.3.2 Configuration .................................................... 3
    1.3.3 Using the HTTP Client API ...................................... 3
  1.4 HTTP server ..................................................... 4
    1.4.1 Introduction ..................................................... 4
    1.4.2 Basic Configuration .............................................. 4
    1.4.3 Server Runtime Configuration ................................... 5
    1.4.4 Server Configuration Directives ............................... 5
    1.4.5 Mime Type Configuration ...................................... 18
    1.4.6 Htaccess - User Configurable Authentication .................. 18
    1.4.7 Dynamic Web Pages ............................................ 20
    1.4.8 Logging .......................................................... 22
    1.4.9 Server Side Includes .......................................... 22
    1.4.10 The Erlang Webserver API ................................... 24
    1.4.11 Inets Webserver Modules ................................... 24
<table>
<thead>
<tr>
<th>2</th>
<th>Inets Reference Manual</th>
<th>29</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>ftp</td>
<td>40</td>
</tr>
<tr>
<td>2.2</td>
<td>http</td>
<td>50</td>
</tr>
<tr>
<td>2.3</td>
<td>http_base_64</td>
<td>54</td>
</tr>
<tr>
<td>2.4</td>
<td>httpd</td>
<td>55</td>
</tr>
<tr>
<td>2.5</td>
<td>httpd_conf</td>
<td>61</td>
</tr>
<tr>
<td>2.6</td>
<td>httpd_socket</td>
<td>63</td>
</tr>
<tr>
<td>2.7</td>
<td>httpd_util</td>
<td>64</td>
</tr>
<tr>
<td>2.8</td>
<td>mod_alias</td>
<td>70</td>
</tr>
<tr>
<td>2.9</td>
<td>mod_auth</td>
<td>72</td>
</tr>
<tr>
<td>2.10</td>
<td>mod_esi</td>
<td>77</td>
</tr>
<tr>
<td>2.11</td>
<td>mod_security</td>
<td>79</td>
</tr>
<tr>
<td>2.12</td>
<td>tftp</td>
<td>82</td>
</tr>
</tbody>
</table>

Glossary

89
Chapter 1

Inets User's Guide

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

1.1 Introduction

1.1.1 Purpose

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

The HTTP client is now documented http(3) [page 50] and supported. It is loosely based on the earlier unsupported client developed by Johan Blom of Mobile Arts AB.

1.1.2 Prerequisites

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

1.1.3 The Service Concept

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

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

```prolog
[Inets, {[services, ListofConfiguredServices]}].
```

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

1.2.1 Introduction

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

1.2.2 Using the FTP Client API

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

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

1.3 HTTP Client

1.3.1 Introduction

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

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

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

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

1.3.3 Using the HTTP Client API

1 > application:start(inets).
ok

Use the proxy "www-proxy.mycompany.com:8000", but not for 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.


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


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


Inets
1.4 HTTP server

1.4.1 Introduction

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

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

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

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

1.4.2 Basic Configuration

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

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

or:

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

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

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

{file,file()} corresponds to the functionality of the old version.
{debug,debug()} is the new trace option. It can trace on all functions or only exported functions on
chosen modules.
{accept_timeout,integer()} sets the wanted timeout value for the server to set up a request connection.
A server config file is specified for each HTTP server to be started. The server config file syntax and
semantics is described in the run time configuration section.
An easy way to test the setup of inets webservers can be done by copying the example server root
(UNIX: $INETS_ROOT/examples/server_root/conf/, Windows: %INETS_ROOT%\examples\server_root\conf\) to a specific installation directory
(/var/tmp/server_root/conf in this example). Then manually start the Erlang node, using inets.config.

    $ erl -config ./inets
    Erlang (BEAM) emulator version 4.9
    Eshell V4.9 (abort with ^G) 1> application:start(inets).
    ok

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

1.4.3 Server Runtime Configuration

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

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

E.g. the syntax is Directive followed by a whitespace followed by the value of the directive followed by a
new line.
The available directives are described in the section Server Configuration Directives.

1.4.4 Server Configuration Directives

Mandatory Directives

- **DIRECTIVE:** “ServerName”
  Syntax: ServerName fully-qualified domain name
  Default: - Mandatory -
  ServerName sets the fully-qualified domain name of the server.

- **DIRECTIVE:** “ServerRoot”
  Syntax: ServerRoot directory-filename
  Default: - Mandatory -
  ServerRoot defines a directory-filename where the server has it's operational home, e.g. used
to store log files and system icons. Relative paths specified in the config file refer to this
directory-filename.
Chapter 1: Inets User’s Guide

- **DIRECTIVE: “DocumentRoot”**
  Syntax: DocumentRoot directory-filename
  Default: - Mandatory -
  DocumentRoot points the Web server to the document space from which to serve documents from. Unless matched by a directive like Alias, the server appends the path from the requested URL to the DocumentRoot to make the path to the document, for example:

  DocumentRoot /usr/web

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

Communication Directives

- **DIRECTIVE: “BindAddress”**
  Syntax: BindAddress address
  Default: BindAddress *
  BindAddress defines which address the server will listen to. If the argument is * then the server listens to all addresses otherwise the server will only listen to the address specified. Address can be given either as an IP address or a hostname.

- **DIRECTIVE: “Port”**
  Syntax: Port number
  Default: Port 80
  Port defines which port number the server should use (0 to 65535). Certain port numbers are reserved for particular protocols, i.e. examine your OS characteristics\(^1\) for a list of reserved ports. The standard port for HTTP is 80.
  All ports numbered below 1024 are reserved for system use and regular (non-root) users cannot use them, i.e. to use port 80 you must start the Erlang node as root. (sic!) If you do not have root access choose an unused port above 1024 typically 8000, 8080 or 8888.

- **DIRECTIVE: “SocketType”**
  Syntax: SocketType type
  Default: SocketType ip_comm
  SocketType defines which underlying communication type to be used. Valid socket types are:
  - ip_comm the default and preferred communication type. ip_comm is also used for all remote message passing in Erlang.
  - ssl the communication type to be used to support SSL.

Limit Directives

- **DIRECTIVE: “DisableChunkedTransferEncodingSend”**
  Syntax: DisableChunkedTransferEncodingSend true | false
  Default: false
  This directive tells the server whether to use chunked transfer-encoding when sending a response to a HTTP/1.1 client.

- **DIRECTIVE: “KeepAlive”**
  Syntax: KeepAlive true | false
  Default: true
  This directive tells the server whether to use persistent connection or not when the client claims to be HTTP/1.1 compliant. Note the value of KeepAlive has changed from previous versions to be compliant with Apache.

\(^1\)In UNIX: /etc/services.
- **DIRECTIVE: “KeepAliveTimeout”**
  Syntax: `KeepAliveTimeout seconds`
  Default: 150
  The number of seconds the server will wait for a subsequent request from the client before closing the connection. If the load on the server is high you may want to shorten this.

- **DIRECTIVE: “MaxBodyAction”**
  Syntax: `MaxBodyAction action`
  Default: `MaxBodyAction close`
  `MaxBodyAction` specifies the action to be taken when the message body limit has been passed.
  
  close  the default and preferred communication type. `ip_comm` is also used for all remote message passing in Erlang.
  
  reply414  a reply (status) message with code 414 will be sent to the client prior to closing the socket. Note that this code is not defined in the HTTP/1.0 version of the protocol.

- **DIRECTIVE: “MaxBodySize”**
  Syntax: `MaxBodySize size`
  Default: `MaxBodySize nolimit`
  `MaxBodySize` limits the size of the message body of HTTP request. The reply to this is specified by the `MaxBodyAction` directive. Valid size is:
  
  nolimit  the default message body limit, e.g. no limit.
  
  integer()  any positive number.

- **DIRECTIVE: “MaxClients”**
  Syntax: `MaxClients number`
  Default: `MaxClients 150`
  `MaxClients` limits the number of simultaneous requests that can be supported. No more than this number of child server processes can be created.

- **DIRECTIVE: “MaxHeaderAction”**
  Syntax: `MaxHeaderAction action`
  Default: `MaxHeaderAction close`
  `MaxHeaderAction` specifies the action to be taken when the message Header limit has been passed.
  
  close  the socket is closed without any message to the client. This is the default action.
  
  reply414  a reply (status) message with code 414 will be sent to the client prior to closing the socket. Note that this code is not defined in the HTTP/1.0 version of the protocol.

- **DIRECTIVE: “MaxHeaderSize”**
  Syntax: `MaxHeaderSize size`
  Default: `MaxHeaderSize 10240`
  `MaxHeaderSize` limits the size of the message header of HTTP request. The reply to this is specified by the `MaxHeaderAction` directive. Valid size is:
  
  integer()  any positive number (default is 10240)
  
  nolimit  no limit should be applied

- **DIRECTIVE: “MaxKeepAliveRequests”**
  Syntax: `MaxKeepAliveRequests NumberOfRequests`
  Default: Disabled
  The number of request that a client can do on one connection. When the server has responded to the number of requests defined by `MaxKeepAliveRequests` the server close the connection. The server will close it even if there are queued request.
DIRECTIVE: "DefaultType"
Syntax: DefaultType mime-type
Default: -None-
When the server is asked to provide a document type which cannot be determined by the MIME Type Settings, the server must inform the client about the content type of documents and mime-type is used if an unknown type is encountered.

DIRECTIVE: "Modules"
Syntax: Modules module module ...
Default: Modules mod_get mod_head mod_log
Modules defines which Erlang Webserver API modules to be used in a specific server setup. module is a module in the code path of the server which has been written in accordance with the section Erlang Web Server API. The server executes functionality in each module, from left to right (from now on called Erlang Webserver API Module Sequence). Before altering the Erlang Webserver API Modules Sequence please observe what types of data each module uses and propagates.

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

    ServerAdmin www-admin@white-house.com

SSL Directives

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

DIRECTIVE: "SSL CertificateFile"
Syntax: SSL CertificateFile filename
Default: -None-
SSL CertificateFile points at a PEM encoded certificate. Read more about PEM encoded certificates in the SSL application documentation. The dummy certificate server.pem, in the Inets distribution, can be used for test purposes. Read more about PEM encoded certificates in the SSL application documentation.

DIRECTIVE: "SSL CertificateKeyFile"
Syntax: SSL CertificateKeyFile filename
Default: -None-
SSL CertificateKeyFile is used to point at a certificate key file. This directive should only be used if a certificate key has not been bundled with the certificate file pointed at by SSL CertificateFile.

DIRECTIVE: "SSLVerifyClient"

2In Windows: %INETS%/examples/server_root/ssl/, in UNIX: $INETS/examples/server_root/ssl/.
Syntax: `SSLVerifyClient` type
Default: - None-

Set type to:

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

Read more about SSL in the application documentation.

**DIRECTIVE: “SSLVerifyDepth”**

Syntax: `SSLVerifyDepth` integer
Default: - None-

This directive specifies how far up or down the (certification) chain we are prepared to go before giving up.

Read more about SSL in the application documentation.

**DIRECTIVE: “SSLCiphers”**

Syntax: `SSLCiphers` ciphers
Default: - None-

`SSLCiphers` is a colon separated list of ciphers.

Read more about SSL in the application documentation.

**DIRECTIVE: “SSLPasswordCallbackFunction”**

Syntax: `SSLPasswordCallbackFunction` function
Default: - None-

The `SSLPasswordCallbackFunction` function in module `SSLPasswordCallbackModule` is called in order to retrieve the user's password.

Read more about SSL in the application documentation.

**DIRECTIVE: “SSLPasswordCallbackModule”**

Syntax: `SSLPasswordCallbackModule` function
Default: - None-

The `SSLPasswordCallbackFunction` function in the `SSLPasswordCallbackModule` module is called in order to retrieve the user's password.

Read more about SSL in the application documentation.

**URL Aliasing**

**DIRECTIVE: “Alias”**

Syntax: `Alias` url-path directory-filename
Default: - None-

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

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

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

**DIRECTIVE: “DirectoryIndex”**

Syntax: `DirectoryIndex` file file ...
Default: - None-
Chapter 1: Inets User's Guide

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

```
DirectoryIndex index.html
```

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

- **DIRECTIVE: “ScriptAlias”**
  Syntax: ScriptAlias url-path directory-filename
  Default: - None -
  The ScriptAlias directive has the same behavior as the Alias directive, except that it also marks the target directory as containing CGI scripts. URLs with a path beginning with url-path are mapped to scripts beginning with directory-filename, for example:

```
ScriptAlias /cgi-bin/ /web/cgi-bin/
```

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

**CGI Directives**

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

```
ScriptNoCache true
```

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

```
ScriptTimeout 15
```

- **DIRECTIVE: “Action”**
  Syntax: Action mime-type cgi-script
  Default: - None -
  Action adds an action, which will activate a cgi-script whenever a file of a certain mime-type is requested. It propagates the URL and file path of the requested document using the standard CGI PATH_INFO and PATH_TRANSLATED environment variables.
  Examples:

```
Action text/plain /cgi-bin/log_and_deliver_text
Action home-grown/mime-type1 ~/bob/do_special_stuff
```
• **DIRECTIVE: “Script”**
  Syntax: Script method cgi-script
  Default: - None -
  Script adds an action, which will activate a cgi-script whenever a file is requested using a certain HTTP method. The method is either GET or POST as defined in RFC 1945. It propagates the URL and file path of the requested document using the standard CGI PATH_INFO and PATH_TRANSLATED environment variables.
  Examples:

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

**ESI Directives**

• **DIRECTIVE: “ErlScriptAlias”**
  Syntax: ErlScriptAlias url-path allowed-module allowed-module ...
  Default: - None -
  ErlScriptAlias marks all URLs matching url-path as erl scheme scripts. A matching URL is mapped into a specific module and function. The module must be one of the allowed-modules.
  For example:

  ```
  ErlScriptAlias /cgi-bin/hit me httpd example md4
  ```

  and a request to http://your.server.org/cgi-bin/hit me/httpd_example:yahoo would refer to httpd_example:yahoo/2.

• **DIRECTIVE: “ErlScriptNoCache”**
  Syntax: ErlScriptNoCache true | false
  Default: false
  If ErlScriptNoCache is set to true the server will add http header fields that prevents proxies from caching the page. This is generally a good idea for dynamic content, since the content often vary between each request.

  ```
  ErlScriptNoCache true
  ```

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

  ```
  ErlScriptTimeout 15
  ```

• **DIRECTIVE: “EvalScriptAlias”**
  Syntax: EvalScriptAlias url-path allowed-module allowed-module ...
  Default: - None -
EvalScriptAlias marks all URLs matching url-path as eval scheme scripts. A matching URL is mapped into a specific module and function. The module must be one of the allowed-modules. For example:

```
EvalScriptAlias /cgi-bin/hit_me_to httpd_example md5
```

And a request to `http://your.server.org/cgi-bin/hit_me_to/httpd_example:print("Hi!")` would refer to `httpd_example:print/1`.

**Auth Directives**

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

  ```
  <Directory /user/local/httpd[12]/htdocs>
  AuthAccessPassword s0M3p3ssW0rd
  AuthDBType plain
  AuthName My Secret Garden
  AuthUserFile /var/tmp/server_root/auth/user
  AuthGroupFile /var/tmp/server_root/auth/group
  require user ragnar edward
  require group group1
  allow from 123.145.244.5
  </Directory>
  ```

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

- **DIRECTIVE:** "AuthDBType"
  - Syntax: `AuthDBType plain | dets | mnesia`
  - Default: - None -
  - Context: Directory
  - `AuthDBType` sets the type of authentication database that is used for the directory. The key difference between the different methods is that dynamic data can be saved when Mnesia and Dets is used.
  - If Mnesia is used as storage method, Mnesia must be started prior to the webserver. The first time Mnesia is started the schema and the tables must be created before Mnesia is started. A naive example of a module with two functions that creates and start mnesia is provided here. The function shall be sued the first time `first_start/0` creates the schema and the tables. The second function `start/0` shall be used in consecutive startups. `start/0` starts Mnesia and wait for the tables to be initiated. This function must only be used when the schema and the tables already is created.

```
-module(mnesia_test).
-export([start/0,load_data/0]).
-include("mod_auth.hrl").
```
First, we define the Mnesia tables:

```erlang
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).
```

Then, we start Mnesia:

```erlang
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 specifies which nodes the database shall reside on. It then starts Mnesia and creates the tables. The first argument is the name of the tables; the second argument is a list of options for how the table will be created, see Mnesia documentation for more information. Since the current implementation of mod_auth_mnesia saves one row for each user, the type must be bag.

When the schema and the tables are created, the second function `start/0` shall be used to start Mnesia. It starts Mnesia and waits for the tables to be loaded. Mnesia uses the directory specified as `mnesia_dir` at startup if specified, otherwise Mnesia uses the current directory.

For security reasons, ensure that the Mnesia tables are stored outside the document tree of the Web server. If it is placed in the directory that it protects, clients will be able to download it.

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

- **DIRECTIVE: "AuthUserFile"**
  - Syntax: `AuthUserFile filename`
  - Default: `None`
  - Context: Directory
  - `AuthUserFile` sets the name of a file which contains the list of users and passwords for user authentication. The file name can be either absolute or relative to `ServerRoot`.

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

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

    If using the `dets` storage method, the user database is maintained by `dets` and should not be edited by hand. Use the API functions in `mod_auth` module to create/edit the user database. This directive is ignored if using the `mnesia` storage method.

    For security reasons, ensure that the `AuthUserFile` is stored outside the document tree of the Web server. If it is placed in the directory that it protects, clients will be able to download it.

- **DIRECTIVE: "AuthGroupFile"**
  - Syntax: `AuthGroupFile filename`
  - Default: `None`
Context: Directory
AuthGroupFile sets the name of a file which contains the list of user groups for user authentication. filename can be either absolute or relative to the ServerRoot.
If you use the plain storage method, the group file is a plain text file, where each line contains a group name followed by a colon, followed by the member user names separated by spaces. For example:

```
group1: bob joe ante
```

If using the dets storage method, the group database is maintained by dets and should not be edited by hand. Use the API for mod_auth module to create / edit the group database.
This directive is ignored if using the mnesia storage method.
For security reasons, make sure that the AuthGroupFile is stored outside the document tree of the Web server. If it is placed in the directory which it protects, clients will be able to download it.

- **DIRECTIVE**: “AuthName”
  Syntax: AuthName auth-domain
  Default: - None -
  Context: Directory
  AuthName sets the name of the authorization realm (auth-domain) for a directory. This string informs the client about which user name and password to use.

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

- **DIRECTIVE**: “allow”
  Syntax: allow from host host ...
  Default: allow from all
  Context: Directory
  allow defines a set of hosts which should be granted access to a given directory. host is one of the following:

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

  For example:

  ```
  allow from 123.34.56.11 150.100.23
  ```

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

- **DIRECTIVE**: “deny”
  Syntax: deny from host host ...
  Default: deny from all
  Context: Directory
  deny defines a set of hosts which should not be granted access to a given directory. host is one of the following:
All hosts are denied access.

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

For example:

```
deny from 123.34.56.11 150.100.23
```

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

- **DIRECTIVE: “require”**
  - Syntax: require entity-name entity entity ...
  - Default: - None -
  - Context: Directory
  - `require` defines users which should be granted access to a given directory using a secret password.
  - The allowed syntaxes are:
    - `require user user-name user-name ...` Only the named users can access the directory.
    - `require group group-name group-name ...` Only users in the named groups can access the directory.

**Htaccess Authentication Directives**

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

**Auth Filter Directives**

- **DIRECTIVE: “SecurityDataFile”**
  - Syntax: SecurityDataFile filename
  - Default: - None -
  - Context: Directory
  - `SecurityDataFile` sets the name of the security modules for a directory. The filename can be either absolute or relative to the `ServerRoot`. This file is used to store persistent data for the `mod_security` module.
  - Several directories can have the same `SecurityDataFile`.

- **DIRECTIVE: “SecurityMaxRetries”**
  - Syntax: SecurityMaxRetries integer() | infinity
  - Default: 3
  - Context: SecurityMaxRetries specifies the maximum number of tries to authenticate a user has before he is blocked out. If a user successfully authenticates when he is blocked, he will receive a 403 (Forbidden) response from the server.
  - For security reasons, failed authentications made by this user will return a message 401 (Unauthorized), even if the user is blocked.

- **DIRECTIVE: “SecurityBlockTime”**
  - Syntax: SecurityBlockTime integer() | infinity
Chapter 1: Inets User's Guide

- **SecurityBlockTime** specifies the number of minutes a user is blocked. After this amount of time, he automatically regains access.
  - *Syntax:* SecurityBlockTime integer() | infinity
  - *Default:* 60
  - *Context:* Directory

- **SecurityFailExpireTime** specifies the number of minutes a failed user authentication is remembered. If a user authenticates after this amount of time, his previous failed authentications are forgotten.
  - *Syntax:* SecurityFailExpireTime integer() | infinity
  - *Default:* 30
  - *Context:* Directory

- **SecurityAuthTimeout** specifies the number of seconds a successful user authentication is remembered. After this time has passed, the authentication will no longer be reported.
  - *Syntax:* SecurityAuthTimeout integer() | infinity
  - *Default:* 30
  - *Context:* Directory

- **SecurityCallbackModule** specifies the name of a callback module.
  - *Syntax:* SecurityCallbackModule atom()
  - *Default:* -None-
  - *Context:* Directory

Logging Directives

- **ErrorLog** defines the filename of the error log file to be used to log server errors. If the filename does not begin with a slash (/) it is assumed to be relative to the ServerRoot, for example:

  ErrorLog logs/error_log_8080

  and errors will be logged in the server root space.

- **SecurityLog** defines the filename of the access log file to be used to log security events. If the filename does not begin with a slash (/) it is assumed to be relative to the ServerRoot. For example:

  SecurityLog logs/security_log_8080

  and security events will be logged in the server root space.

- **TransferLog**
  - *Syntax:* TransferLog filename
  - *Default:* -None-

---

3In Windows: %SERVER\ROOT%\logs\error_log_8080. In UNIX: $SERVER_ROOT/logs/error_log_8080.
TransferLog defines the filename of the access log file to be used to log incoming requests. If the filename does not begin with a slash (/) it is assumed to be relative to the ServerRoot. For example:

```
TransferLog logs/access_log_8080
```

and errors will be logged in the server root\(^5\) space.

**Disk Log Directives**

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

```
DiskLogFormat external
```

- **DIRECTIVE**: “ErrorDiskLog”
  Syntax: ErrorDiskLog filename
  Default: - None -
  ErrorDiskLog defines the filename of the (disk_log(3)) error log file to be used to log server errors. If the filename does not begin with a slash (/) it is assumed to be relative to the ServerRoot, for example:

```
ErrorDiskLog logs/error_disk_log_8080
```

and errors will be logged in the server root\(^6\) space.

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

- **DIRECTIVE**: “SecurityDiskLog”
  Syntax: SecurityDiskLog filename
  Default: - None -
  SecurityDiskLog defines the filename of the (disk_log(3)) access log file which logs incoming security events i.e authenticated requests. If the filename does not begin with a slash (/) it is assumed to be relative to the ServerRoot.

- **DIRECTIVE**: “SecurityDiskLogSize”
  Syntax: SecurityDiskLogSize max-bytes max-files
  Default: SecurityDiskLogSize 512000 8

\(^5\)In Windows: %SERVER_ROOT%\logs\access_log_8080. In UNIX: $SERVER_ROOT/logs/access_log_8080.

\(^6\)In Windows: %SERVER_ROOT%\logs\error_disk_log_8080. In UNIX: $SERVER_ROOT/logs/error_disk_log_8080.
Chapter 1: Inets User's Guide

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

- **DIRECTIVE: “TransferDiskLog”**
  Syntax: TransferDiskLog filename
  Default: - None -
  TransferDiskLog defines the filename of the (disk_log(3)) access log file which logs incoming requests. If the filename does not begin with a slash (/) it is assumed to be relative to the ServerRoot, for example:

  TransferDiskLog logs/transfer_disk_log_8080

  and errors will be logged in the server root space.

- **DIRECTIVE: “TransferDiskLogSize”**
  Syntax: TransferDiskLogSize max-bytes max-files
  Default: TransferDiskLogSize 512000 8
  TransferDiskLogSize defines the properties of the disk_log(3) access log file. The disk_log(3) access log file is of type wrap log and max-bytes will be written to each file and max-files will be used before the first file is truncated and reused.

### 1.4.5 Mime Type Configuration

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

The mapping between file suffixes and MIME types are specified in the mime.types file. The mime.types reside within the conf directory of the ServerRoot. MIME types may be added as required to the mime.types file and the DefaultType config directive can be used to specify a default mime type.

An example of a very small mime.types file:

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

### 1.4.6 Htaccess - User Configurable Authentication.

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

- Each directory in the path to the requested asset is searched for an access-file (default .htaccess), that restricts the webservers rights to respond to a request. If an access-file is found the rules in that file are 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.

---

7 In Windows: %SERVER_ROOT%\logs\transfer_disk_log_8080. In UNIX: $SERVER_ROOT/logs/transfer_disk_log_8080.
All the access-files in the path to a requested asset is read once per request, this means that the load on the server will increase when this scheme is used.

If a directory is limited both by auth directives in the HTTP server configuration file and by the .htaccess files. The user must be allowed to get access the file by both methods for the request to succeed.

Access Files Directives

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

- DIRECTIVE: “allow” Syntax: Allow from subnet subnet from all
  Default: from all
  Same as the directive allow for the server config file.

- DIRECTIVE: “AllowOveRide” Syntax: AllowOveRide all | none | Directives
  Default: - None -
  AllowOveRide Specify which parameters that not access-files in subdirectories are allowed to alter the value for. If the parameter is set to none no more access-files will be parsed.
  If only one access-file exists setting this parameter to none can lessen the burden on the server since the server will stop looking for access-files.

- DIRECTIVE: “AuthGroupfile” Syntax: AuthGroupFile Filename
  Default: - None -
  AuthGroupFile indicates which file that contains the list of groups. Filename must contain the absolute path to the file. The format of the file is one group per row and every row contains the name of the group and the members of the group separated by a space, for example:

  GroupName: Member1 Member2 .... MemberN

- DIRECTIVE: “AuthName” Syntax: AuthName auth-domain
  Default: - None -
  Same as the directive AuthName for the server config file.

- DIRECTIVE: “AuthType” Syntax: AuthType Basic
  Default: Basic
  AuthType Specify which authentication scheme that shall be used. Today only Basic Authenticating using UUEncoding of the password and user ID is implemented.

- DIRECTIVE: “AuthUserFile” Syntax: AuthUserFile Filename
  Default: - None -
  AuthUserFile indicate which file that contains the list of users. Filename must contain the absolute path to the file. The users name and password are not encrypted so do not place the file with users in a directory that is accessible via the webserver. The format of the file is one user per row and every row contains UserName and Password separated by a colon, for example:

  UserName:Password
  UserName:Password

- DIRECTIVE: “deny” Syntax: deny from subnet subnet from all
  Context: Limit
Chapter 1: Inets User's Guide

Same as the directive deny for the server config file.

- **DIRECTIVE: “Limit”**
  
  Syntax: `<Limit RequestMethods>`
  
  Default: `-None-
  
  `<Limit>` and `</Limit>` are used to enclose a group of directives which applies only to requests using the specified methods. If no request method is specified all request methods are verified against the restrictions.

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

- **DIRECTIVE: “order”**
  
  Syntax: `order allow deny | deny allow`
  
  Default: `allow deny`
  
  `order`, defines if the deny or allow control shall be preformed first.
  
  If the order is set to allow deny, then first the users network address is controlled to be in the allow subset. If the users network address is not in the allowed subset he will be denied to get the asset. If the network-address is in the allowed subset then a second control will be preformed, that the users network address is not in the subset of network addresses that shall be denied as specified by the deny parameter.
  
  If the order is set to deny allow then only users from networks specified to be in the allowed subset will succeed to request assets in the limited area.

- **DIRECTIVE: “require”**
  
  Syntax: `require group group1 group2... | user user1 user2...`
  
  Default: `-None-
  
  Context: `Limit`
  
  See the require directive in the documentation of mod_auth(3) for more information.

1.4.7 Dynamic Web Pages

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

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

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

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

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

The CGI Script response comprises a message-header and a message-body, separated by a blank line. The message-header contains one or more header fields. The body may be empty. Example:
The server will interpret the cgi-headers and most of them will be transformed into HTTP headers and sent back to the client together with the body.
Support for CGI-1.1 is implemented in accordance with the RFC 3875.

Erlang Server Interface (ESI)
The erlang server interface is implemented by the module mod esi.

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

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

*** above depends on how the ErlScriptAlias config directive has been used
The module (Module) referred to must be found in the code path, and it must define a function (Function) with an arity of two or three. It is preferable to implement a function with arity three as it permits you to send chunks of the webpage being 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 backward compatibility reasons. See mod esi(3) [page 77] for implementation details of the esi callback function.

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

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

*** above depends on how the ErlScriptAlias config directive has been used
The module (Mod) referred to must be found in the code path, and data returned by the function (Func) is passed back to the client. Data returned from the function must furthermore take the form as specified in the CGI specification. See mod esi(3) [page 77] for implementation details of the esi callback function.

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

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

which effectively will close down the Erlang node, that is use the erl scheme instead, until this security breach has been fixed.
Today there are no good way of solving this problem and therefore Eval Scheme may be removed in future release of Inets.
1.4.8 Logging

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

remotehost rfc931 authuser [date] “request” status bytes

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

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

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

1.4.9 Server Side Includes

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

**Note:**

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

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

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

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

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

```
text/x-server-parsed-html html htm
```
Server-Side Includes (SSI) Format

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

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

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

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

**include** will insert the text of a document into the parsed document. This command accepts two tags:
- **virtual** gives a virtual path to a document on the server. Only normal files and other parsed documents can be accessed in this way.
- **file** gives a pathname relative to the current directory. ../ cannot be used in this pathname, nor can absolute paths. As above, you can send other parsed documents, but you cannot send CGI scripts.

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

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

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

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

Server-Side Includes (SSI) Environment Variables

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

- DOCUMENT_NAME The current filename.
- DOCUMENT_URI The virtual path to this document (such as /docs/tutorials/foo.shtml).
- QUERY_STRING_UNESCAPED The unescaped version of any search query the client sent, with all shell-special characters escaped with \.
- DATE_LOCAL The current date, local time zone.
- DATE_GMT Same as DATE_LOCAL but in Greenwich mean time.
- LAST_MODIFIED The last modification date of the current document.
1.4.10 The Erlang Webserver API

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

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

To provide customization and extensibility of the 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 Webserver API. A description of functionality implemented by the Erlang Webserver API is described in the section Inets Webserver Modules.

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

**API Description**

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

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

The latter functions are needed only when new config directives are to be introduced. For details see httpd(3) [page 55]

1.4.11 Inets Webserver Modules

The convention is that all modules implementing some webserver functionality has the name mod_*_. When configuring the webserver 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 Webserver API interaction data, if available:
- real_name - from mod_alias
Exports the following Erlang Webserver API interaction data, if possible:
{new_request_uri, RequestURI} An alternative RequestURI has been generated.

mod_alias - URL Aliasing
This module makes it possible to map different parts of the host file system into the document tree e.i. creates aliases and redirections.
Exports the following Erlang Webserver API interaction data, if possible:
{real_name, PathData} PathData is the argument used for API function mod_alias:path/3.

mod_auth - User Authentication
This module provides for basic user authentication using textual files, dets databases as well as mnesia databases.
Uses the following Erlang Webserver API interaction data, if available:
- real_name - from mod_alias
Exports the following Erlang Webserver API interaction data, if possible:
{remote_user, User} The user name with which the user has authenticated himself.

mod cgi - CGI Scripts
This module handles invoking of CGI scripts

mod_dir - Directories
This module generates an HTML directory listing (Apache-style) if a client sends a request for a directory instead of a file. This module needs to be removed from the Modules config directive if directory listings is unwanted.
Uses the following Erlang Webserver API interaction data, if available:
- real_name - from mod_alias
Exports the following Erlang Webserver API interaction data, if possible:
{mime_type, MimeType} The file suffix of the incoming URL mapped into a MimeType.
mod_disk_log - Logging Using disk_log.

Standard logging using the "Common Logfile Format" and disk_log(3).
Uses the following Erlang Webserver API interaction data, if available:
- remote_user - from mod_auth

mod_esi - Erlang Server Interface

This module implements the Erlang Server Interface (ESI) that provides a tight and efficient interface
to the execution of Erlang functions.
Uses the following Erlang Webserver API interaction data, if available:
- remote_user - from mod_auth

Exports the following Erlang Webserver API interaction data, if possible:
- {mime_type, MimeType} The file suffix of the incoming URL mapped into a MimeType

mod_get - Regular GET Requests

This module is responsible for handling GET requests to regular files. GET requests for parts of files is
handled by mod_range.
Uses the following Erlang Webserver API interaction data, if available:
- real_name - from mod_alias

mod_head - Regular HEAD Requests

This module is responsible for handling HEAD requests to regular files. HEAD requests for dynamic
content is handled by each module responsible for dynamic content.
Uses the following Erlang Webserver API interaction data, if available:
- real_name - from mod_alias

mod_htaccess - User Configurable Access

This module provides per-directory user configurable access control.
Uses the following Erlang Webserver API interaction data, if available:
- real_name - from mod_alias

Exports the following Erlang Webserver API interaction data, if possible:
- {remote_user_name, User} The user name with which the user has authenticated himself.
mod\_include - SSI

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

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

- real\_name - from mod\_alias
- remote\_user - from mod\_auth

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

\{mime\_type, MimeType\} The file suffix of the incoming URL mapped into a Mime\_Type as defined in the Mime Type Settings section.

mod\_log - Logging Using Text Files.

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

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

- remote\_user - from mod\_auth

mod\_range - Requests with Range Headers

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

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

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

- real\_name - from mod\_alias

mod\_response\_control - Requests with If* Headers

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

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

1. If-modified
2. If-Unmodified
3. If-Match
4. If-\text{Not}match

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

- real\_name - from mod\_alias

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

\{if\_range, send\_file\} The conditions for the range request was not fullfilled. The response must not be treated as a range request, instead it must be treated as a ordinary get request.
mod_security - Security Filter

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

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

mod_trace - TRACE Request

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

Short Summaries

- Erlang Module `ftp` [page 40] – A File Transfer Protocol client
- Erlang Module `http` [page 50] – An HTTP/1.1 client
- Erlang Module `http_base64` [page 54] – Implements base 64 encode and decode, see RFC 2045.
- Erlang Module `httpd` [page 55] – An implementation of an HTTP 1.1 compliant Web server, as defined in RFC 2616.
- Erlang Module `httpd_conf` [page 61] – Configuration utility functions to be used by the Erlang Webserver API programmer.
- Erlang Module `httpd_socket` [page 63] – Communication utility functions to be used by the Erlang Webserver API programmer.
- Erlang Module `httpd_util` [page 64] – Miscellaneous utility functions to be used when implementing Erlang Webserver API modules.
- Erlang Module `mod_auth` [page 72] – User authentication using text files, dets or mnesia database.
- Erlang Module `mod_esi` [page 77] – Erlang Server Interface
- Erlang Module `mod_security` [page 79] – Security Audit and Trailing Functionality
- Erlang Module `tftp` [page 82] – Trivial FTP

### ftp

The following functions are exported:

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

- `append_chunk_end(Pid) -> ok | {error, Reason}`
  
  [page 42] Stop transfer of chunks for appending.

- `cd(Pid, Dir) -> ok | {error, Reason}`
  
  [page 42] Change remote working directory.

- `close(Pid) -> ok`  
  
  [page 42] End ftp session.

- `delete(Pid, File) -> ok | {error, Reason}`
  
  [page 42] Delete a file at the remote server.

- `formaterror(Tag) -> string()`
  
  [page 42] Return error diagnostics.

- `lcd(Pid, Dir) -> ok | {error, Reason}`
  
  [page 42] Change local working directory.

- `lpwd(Pid) -> ok, Dir`  
  
  [page 45] Get local current working directory.

- `ls(Pid [, Dir]) -> {ok, Listing} | {error, Reason}`
  
  [page 43] List contents of remote directory.

- `mkdir(Pid, Dir) -> ok | {error, Reason}`
  
  [page 43] Create remote directory.

- `nlist(Pid [, Dir]) -> {ok, Listing} | {error, Reason}`
  
  [page 43] List contents of remote directory.

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

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

- `pwd(Pid) -> {ok, Dir} | {error, Reason}`
  
  [page 45] Get remote current working directory.

- `recv(Pid, RemoteFile [, LocalFile]) -> ok | {error, Reason}`
  
  [page 45] Transfer file from remote server.

- `recv_bin(Pid, RemoteFile) -> {ok, Bin} | {error, Reason}`
  
  [page 47] Transfer a binary into a remote file.

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

- `recv_chunk(Pid) -> ok | {ok, Bin} | {error, Reason}`
  
  [page 46] Write a chunk to the remote file.

- `rename(Pid, Old, New) -> ok | {error, Reason}`
  
  [page 46] Rename a file at the remote server.

- `rmdir(Pid, Dir) -> ok | {error, Reason}`
  
  [page 46] Remove a remote directory.

- `send(Pid, LocalFile [, RemoteFile]) -> ok | {error, Reason}`
  
  [page 46] Transfer file to remote server.

- `send_bin(Pid, Bin, RemoteFile) -> ok | {error, Reason}`
  
  [page 47] Transfer a binary into a remote file.

- `send_chunk(Pid, Bin) -> ok | {error, Reason}`
  
  [page 47] Write a chunk to the remote file.

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

**http**

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

**http_base_64**

The following functions are exported:
- **encode(PlainASCII) -> Base64**  
  [page 54] Encodes a plain ASCII string into base64.
- **decode(Base64) -> PlainASCII**  
  [page 54] Decodes an base64 encoded string to plain ASCII.
httpd

The following functions are exported:

- **Module:do(ModData) -**
  - {proceed, OldData} | {proceed, NewData} | {break, NewData} | done
  - [page 56] Called for each request to the Web server.

- **Module:load(Line, Context) -**
  - eof | ok | {ok, NewContext} | {ok, NewContext, Directive} | {ok, NewContext, DirectiveList} | {error, Reason}
  - [page 57] Load a configuration directive.

  - {ok, DirectiveKey, NewDirectiveValue} | {ok, DirectiveKey, NewDirectiveValue} | {error, Reason}
  - [page 57] Alter the value of one or more configuration directive.

- **Module:remove(ConfigDB) -**
  - ok | {error, Reason}
  - [page 57] Callback function that is called when the Web server is closed.

- **parse_query(QueryString) -**
  - ServerRet
  - [page 58] Parse incoming data to erl and eval scripts

- **start() -**
  - [page 58] Start a server as specified in the given config file.

- **start(Config) -**
  - ServerRet
  - [page 58] Start a server as specified in the given config file.

- **start_link() -**
  - [page 58] Start a server as specified in the given config file.

- **start_link(Config) -**
  - ServerRet
  - [page 58] Start a server as specified in the given config file.

- **restart() -**
  - [page 59] Restart a running server.

- **restart(Port) -**
  - ok | {error, Reason}
  - [page 59] Restart a running server.

- **restart(ConfigFile) -**
  - ServerRet
  - [page 59] Restart a running server.

- **restart(Address,Port) -**
  - ServerRet
  - [page 59] Restart a running server.

- **stop() -**
  - [page 59] Stop a running server.

- **stop(Port) -**
  - ServerRet
  - [page 59] Stop a running server.

- **stop(ConfigFile) -**
  - ServerRet
  - [page 59] Stop a running server.

- **stop(Address,Port) -**
  - ServerRet
  - [page 59] Stop a running server.

- **block() -**
  - ok | {error,Reason}
  - [page 59] Block a running server.

- **block(Port) -**
  - ok | {error,Reason}
  - [page 59] Block a running server.
The following functions are exported:

- **check** (EnumString, ValidEnumStrings) -> Result
  [page 61] Check if string is a valid enumeration.

- **clean** (String) -> Stripped
  [page 61] Remove leading and/or trailing white spaces.

- **custom_clean** (String, Before, After) -> Stripped
  [page 61] Remove leading and/or trailing white spaces and custom characters.

- **is_directory** (FilePath) -> Result
  [page 61] Check if a file path is a directory.

- **is_file** (FilePath) -> Result
  [page 62] Check if a file path is a regular file.

- **make_integer** (String) -> Result
  [page 62] Return an integer representation of a string.

The following functions are exported:

- **deliver** (SocketType, Socket, Data) -> Result
  [page 63] Send binary data over socket.

- **peername** (SocketType, Socket) -> {Port, IPAddress}
  [page 63] Return the port and IP-address of the remote socket.

- **resolve** () -> HostName
  [page 63] Return the official name of the current host.
The following functions are exported:

- `convert_request_date(DateString) -> ErlDate|bad_date`
  [page 64] Convert the date to the Erlang date format.
- `create_etag(FileInfo) -> Etag`
  [page 64] Calculates the Etag for a file.
- `decode_base64(Base64String) -> ASCIIString`
  [page 64] Convert a base64 encoded string to a plain ascii string.
- `decode_hex(HexValue) -> DecValue`
  [page 64] Convert a hex value into its decimal equivalent.
- `day(NthDayOfWeek) -> DayOfWeek`
  [page 64] Convert the day of the week (integer [1-7]) to an abbreviated string.
- `encode_base64(ASCIIString) -> Base64String`
  [page 65] Convert an ASCII string to a Base64 encoded string.
- `flatlength(NestedList) -> Size`
  [page 65] Compute the size of a possibly nested list.
- `header(StatusCode,PersistentConn)`
  [page 65] Generate a HTTP 1.1 header.
- `header(StatusCode,Date)`
  [page 65] Generate a HTTP 1.1 header.
- `header(StatusCode,MimeType,Date)`
  [page 65] Generate a HTTP 1.1 header.
- `header(StatusCode,MimeType,PersistentConn,Date) -> HTTPHeader`
  [page 65] Generate a HTTP 1.1 header.
- `hexlist_to_integer(HexString) -> Number`
  [page 65] Convert a hexadecimal string to an integer.
- `integer_tohexlist(Number) -> HexString`
  [page 65] Convert an integer to a hexadecimal string.
- `key1search(TupleList,Key)`
  [page 66] Search a list of key-value tuples for a tuple whose first element is a key.
- `key1search(TupleList,Key,Undefined) -> Result`
  [page 66] Search a list of key-value tuples for a tuple whose first element is a key.
- `lookup(ETSTable,Key) -> Result`
  [page 66] Extract the first value associated with a key in an ETS table.
- `lookup(ETSTable,Key,Undefined) -> Result`
  [page 66] Extract the first value associated with a key in an ETS table.
- `lookup_mime(ConfigDB,Suffix)`
  [page 66] Return the mime type associated with a specific file suffix.
- `lookup_mime(ConfigDB,Suffix,Undefined) -> MimeType`
  [page 66] Return the mime type associated with a specific file suffix.
- `lookup_mime_default(ConfigDB,Suffix)`
  [page 66] Return the mime type associated with a specific file suffix or the value of the DefaultType.
- `lookup_mime_default(ConfigDB,Suffix,Undefined) -> MimeType`
  [page 66] Return the mime type associated with a specific file suffix or the value of the DefaultType.
• `message(StatusCode, PhraseArgs, ConfigDB) -> Message`  
  [page 67] Return an informative HTTP 1.1 status string in HTML.

• `month(NthMonth) -> Month`  
  [page 67] Convert the month as an integer (1-12) to an abbreviated string.

• `multi_lookup(ETSTable, Key) -> Result`  
  [page 67] Extract the values associated with a key in a ETS table.

• `reason_phrase(StatusCode) -> Description`  
  [page 67] Return the description of an HTTP 1.1 status code.

• `rfc1123_date() -> RFC1123Date`  
  [page 68] Return the current date in RFC 1123 format.

• `rfc1123_date({YYYY, MM, DD}, {Hour, Min, Sec}) -> RFC1123Date`  
  [page 68] Return the current date in RFC 1123 format.

• `split(String, RegExp, N) -> SplitRes`  
  [page 68] Split a string in N chunks using a regular expression.

• `split_script_path(RequestLine) -> Splitted`  
  [page 68] Split a RequestLine in a file reference to an executable and a QueryString or a PathInfo String.

• `split_path(RequestLine) -> {Path, QueryStringOrPathInfo}`  
  [page 68] Split a RequestLine in a file reference and a QueryString or a PathInfo String.

• `strip(String) -> Stripped`  
  [page 68] Returns String where the leading and trailing space and tabs has been removed.

• `suffix(FileName) -> Suffix`  
  [page 69] Extract the file suffix from a given filename.

• `to_lower(String) -> ConvertedString`  
  [page 69] Convert upper-case letters to lower-case.

• `to_upper(String) -> ConvertedString`  
  [page 69] Convert lower-case letters to upper-case.

**mod_alias**

The following functions are exported:

• `default_index(ConfigDB, Path) -> NewPath`  
  [page 70] Return a new path with the default resource or file appended.

• `path(PathData, ConfigDB, RequestURI) -> Path`  
  [page 70] Return the actual file path to a URL.

• `real_name(ConfigDB, RequestURI, Aliases) -> Ret`  
  [page 70] Expand a request uri using Alias config directives.

• `real_script_name(ConfigDB, RequestURI, ScriptAliases) -> Ret`  
  [page 71] Expand a request uri using ScriptAlias config directives.
mod_auth

The following functions are exported:

- `add_user(UserName, Options) -> true | {error, Reason}` [page 72] Add a user to the user database.
- `add_user(UserName, Password, UserData, Port, Dir) -> true | {error, Reason}` [page 72] Add a user to the user database.
- `add_user(UserName, Password, UserData, Address, Port, Dir) -> true | {error, Reason}` [page 72] Add a user to the user database.
- `delete_user(UserName, Options) -> true | {error, Reason}` [page 72] Delete a user from the user database.
- `delete_user(UserName, Port, Dir) -> true | {error, Reason}` [page 72] Delete a user from the user database.
- `delete_user(UserName, Address, Port, Dir) -> true | {error, Reason}` [page 72] Delete a user from the user database.
- `get_user(UserName, Options) -> {ok, #httpd_user} | {error, Reason}` [page 73] Returns a user from the user database.
- `get_user(UserName, Port, Dir) -> {ok, #httpd_user} | {error, Reason}` [page 73] Returns a user from the user database.
- `get_user(UserName, Address, Port, Dir) -> {ok, #httpd_user} | {error, Reason}` [page 73] Returns a user from the user database.
- `list_users(Options) -> {ok, Users} | {error, Reason}` [page 73] List users in the user database.
- `list_users(Port, Dir) -> {ok, Users} | {error, Reason}` [page 73] List users in the user database.
- `list_users(Address, Port, Dir) -> {ok, Users} | {error, Reason}` [page 73] List users in the user database.
- `add_group_member(GroupName, UserName, Options) -> true | {error, Reason}` [page 73] Add a user to a group.
- `add_group_member(GroupName, UserName, Port, Dir) -> true | {error, Reason}` [page 73] Add a user to a group.
- `add_group_member(GroupName, UserName, Address, Port, Dir) -> true | {error, Reason}` [page 73] Add a user to a group.
- `delete_group_member(GroupName, UserName, Options) -> true | {error, Reason}` [page 74] Remove a user from a group.
- `delete_group_member(GroupName, UserName, Port, Dir) -> true | {error, Reason}` [page 74] Remove a user from a group.
- `delete_group_member(GroupName, UserName, Address, Port, Dir) -> true | {error, Reason}` [page 74] Remove a user from a group.
- `list_group_members(GroupName, Options) -> {ok, Users} | {error, Reason}`  
  [page 74] List the members of a group.
- `list_group_members(GroupName, Port, Dir) -> {ok, Users} | {error, Reason}`  
  [page 74] List the members of a group.
- `list_group_members(GroupName, Address, Port, Dir) -> {ok, Users} | {error, Reason}`  
  [page 74] List the members of a group.
- `list_groups(Options) -> {ok, Groups} | {error, Reason}`  
  [page 75] List all the groups.
- `list_groups(Port, Dir) -> {ok, Groups} | {error, Reason}`  
  [page 75] List all the groups.
- `list_groups(Address, Port, Dir) -> {ok, Groups} | {error, Reason}`  
  [page 75] List all the groups.
- `delete_group(GroupName, Options) -> true | {error, Reason}`  
  [page 75] Deletes a group.
- `<name>delete_group(GroupName, Port, Dir) -> true | {error, Reason}`  
  [page 75] Deletes a group.
- `delete_group(GroupName, Address, Port, Dir) -> true | {error, Reason}`  
  [page 75] Deletes a group.
- `update_password(Port, Dir, OldPassword, NewPassword, NewPassword) -> ok | {error, Reason}`  
  [page 75] Change the AuthAcessPassword
- `update_password(Address, Port, Dir, OldPassword, NewPassword, NewPassword) -> ok | {error, Reason}`  
  [page 75] Change the AuthAcessPassword

**mod_esi**

The following functions are exported:

- `deliver(SessionID, Data) -> ok | {error, Reason}`  
  [page 77] Sends Data back to client.
- `Module:Function(SessionID, Env, Input) -> _`  
  [page 77] Creates a dynamic web page and returns it chunk by chunk to the server process by calling mod_esi:deliver/2.
- `Module:Function(Env, Input) -> Response`  
  [page 78] Creates a dynamic web page and return it as a list. This functions is deprecated and only kept for backwards compability.

**mod_security**

The following functions are exported:

- `list_auth_users(Port) -> Users | []`  
  [page 79] List users that have authenticated within the SecurityAuthTimeout time for a given address (if specified), port number and directory (if specified).
Inets Reference Manual

- `list_auth_users(Address, Port) -> Users | []`
  [page 79] List users that have authenticated within the SecurityAuthTimeout time for a given address (if specified), port number and directory (if specified).

- `list_auth_users(Port, Dir) -> Users | []`
  [page 79] List users that have authenticated within the SecurityAuthTimeout time for a given address (if specified), port number and directory (if specified).

- `list_auth_users(Address, Port, Dir) -> Users | []`
  [page 79] List users that have authenticated within the SecurityAuthTimeout time for a given address (if specified), port number and directory (if specified).

- `list_blocked_users(Port) -> Users | []`
  [page 79] List users that are currently blocked from access to a specified port number, for a given address (if specified).

- `list_blocked_users(Address, Port) -> Users | []`
  [page 79] List users that are currently blocked from access to a specified port number, for a given address (if specified).

- `list_blocked_users(Port, Dir) -> Users | []`
  [page 79] List users that are currently blocked from access to a specified port number, for a given address (if specified).

- `list_blocked_users(Address, Port, Dir) -> Users | []`
  [page 79] List users that are currently blocked from access to a specified port number, for a given address (if specified).

- `block_user(User, Port, Dir, Seconds) -> true | {error, Reason}`
  [page 79] Block user from access to a directory for a certain amount of time.

- `block_user(User, Address, Port, Dir, Seconds) -> true | {error, Reason}`
  [page 79] Block user from access to a directory for a certain amount of time.

- `unblock_user(User, Port) -> true | {error, Reason}`
  [page 80] Remove a blocked user from the block list.

- `unblock_user(User, Address, Port) -> true | {error, Reason}`
  [page 80] Remove a blocked user from the block list.

- `unblock_user(User, Port, Dir) -> true | {error, Reason}`
  [page 80] Remove a blocked user from the block list.

- `unblock_user(User, Address, Port, Dir) -> true | {error, Reason}`
  [page 80] Remove a blocked user from the block list.

- `event(What, Port, Dir, Data) -> ignored`
  [page 80] This function is called whenever an event occurs in mod_security.

- `event(What, Address, Port, Dir, Data) -> ignored`
  [page 80] This function is called whenever an event occurs in mod_security.

**tftp**

The following functions are exported:

- `start(Options) -> {ok, Pid} | {error, Reason}`
  [page 83] Start a daemon process.

- `read_file(RemoteFilename, LocalFilename, Options) -> {ok, LastCallbackState} | {error, Reason}`
  [page 84] Read a (virtual) file from a TFTP server.
- `write_file(RemoteFilename, LocalFilename, Options) -> {ok, LastCallbackState} | {error, Reason}`
  [page 84] Write a (virtual) file to a TFTP server
- `info(Pid) -> undefined | Options`
  [page 84] Return information about a daemon, server or client process
- `start() -> ok | {error, Reason}`
  [page 84] Start the Inets application
- `prepare(Access, Filename, Mode, SuggestedOptions, InitialState) -> {ok, AcceptedOptions, NewState} | {error, {Code, Text}}`
  [page 85] Prepare to open a file on the client side
- `open(Access, Filename, Mode, SuggestedOptions, State) -> {ok, AcceptedOptions, NewState} | {error, {Code, Text}}`
  [page 86] Open a file for read or write access
- `read(State) -> {more, Bin, NewState} | {last, Bin, FileSize} | {error, {Code, Text}}`
  [page 86] Read a chunk from the file
- `write(Bin, State) -> {more, NewState} | {last, FileSize} | {error, {Code, Text}}`
  [page 86] Write a chunk to the file
- `abort(Code, Text, State) -> ok`
  [page 87] Abort the file transfer
The ftp module implements a client for file transfer according to a subset of the File Transfer Protocol (see RFC 959). Starting from inets version 4.4.1 the ftp client will always try to use passive ftp mode and only resort to active ftp mode if this fails. There is a mode option for `open/[1,2,3]` where this default behavior may be changed. Also the mode option replaces the API function `force_active/1` that now has become deprecated. (`force_active/1` is removed from the documentation but is still available in the code for now.)

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

In addition to the ordinary functions for receiving and sending files (see `recv/2`, `recv/3`, `send/2` and `send/3`) there are functions for receiving remote files as binaries (see `recv_bin/2`) and for sending binaries to to be stored as remote files (see `send_bin/3`). There is also a set of functions for sending and receiving contiguous parts of a file to be stored in a remote file (for send see `send_chunk_start/2`, `send_chunk/2` and `send_chunk_end/1` and for receive see `recv_chunk_start/2` and `recv_chunk/`).

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

**COMMON DATA TYPES**

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

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

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

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

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

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

Types:
- Pid = pid()
- LocalFile = RemoteFile = string()
- Reason = epath | elogin | etnospc | epnospc | efnamea | 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 exist, 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 exist, the file 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 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 \texttt{File} at the remote server. If the file does not exist it will be created.

\begin{verbatim}
append_chunk_end(Pid) -> ok | {error, Reason}
   Types:
   * Pid = pid()
   * Reason = echunk | restriction_reason() | shortage_reason()
\end{verbatim}

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

\begin{verbatim}
cd(Pid, Dir) -> ok | {error, Reason}
   Types:
   * Pid = pid()
   * Dir = string()
   * Reason = restriction_reason() | common_reason()
\end{verbatim}

Changes the working directory at the remote server to \texttt{Dir}.

\begin{verbatim}
close(Pid) -> ok
   Types:
   * Pid = pid()
\end{verbatim}

Ends the ftp session.

\begin{verbatim}
delete(Pid, File) -> ok | {error, Reason}
   Types:
   * Pid = pid()
   * File = string()
   * Reason = restriction_reason() | common_reason()
\end{verbatim}

Deletes the file \texttt{File} at the remote server.

\begin{verbatim}
formaterror(Tag) -> string()
   Types:
   * Tag = \{error, atom()\} | atom()
\end{verbatim}

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

\begin{verbatim}
lcd(Pid, Dir) -> ok | {error, Reason}
   Types:
   * Pid = pid()
   * Dir = string()
   * Reason = restriction_reason()
\end{verbatim}

Changes the working directory to \texttt{Dir} for the local client.

\begin{verbatim}
lpwd(Pid) -> {ok, Dir}
\end{verbatim}
Types:
- Pid = pid()

Returns the current working directory at the local client.

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

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

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

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

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

Creates the directory Dir at the remote server.

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

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

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

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

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

Types:
- Host = string() \mid ip_address()
- \text{ip_address}() = \{byte(), byte(), byte(), byte()\} \{byte(), byte(), byte(), byte(), byte(), byte(), byte(), byte(), byte()\}
- byte() = 0 \mid 1 \mid ... \mid 255
- Port = integer()
- Flags = [Flag]
- Flag = verbose \mid debug \mid ip_v6_disabled
- Pid = pid()
Reason = ehost
Option_list = [Options]
Options = {host, Host} | {port, Port} | {mode, Mode} | {flags, Flags} | {timeout, Timeout} | {progress, ProgressOption}
Mode = active | passive
Timeout = integer()
ProgressOption = ignore | {CBModule, CBFunction, InitProgress}
CallBackModule = atom()
CallBackFunction = atom()
InitProgress = term()

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

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

Default value for Mode is passive.

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

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

- Before a file is transferred the following call will be made to indicate the start of the file transfer and how big the file is. The return value of the callback function should be a new value for the UserProgressTerm that will be used as input next time the callback function is called.
  
  CBModule:CBFunction(InitProgress, File, {file_size, FileSize})

- Every time a chunk of bytes is transferred the following call will be made:
  
  CBModule:CBFunction(UserProgressTerm, File, {transfer_size, TransferSize})

- At the end of the file the following call will be made to indicate the end of the transfer.
  
  CBModule:CBFunction(UserProgressTerm, File, {transfer_size, 0})

The callback function should be defined as

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

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

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

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

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

The timeout value is default set to 60000 milliseconds.

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

\[ \text{pwd}(\text{Pid}) \rightarrow \{\text{ok, Dir} \} | \{\text{error, Reason} \} \]

Types:
- `Pid = \text{pid}()`
- `Reason = \text{restriction_reason()} | \text{common_reason()}`

Returns the current working directory at the remote server.

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

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

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

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

\[ \text{recv_bin}(\text{Pid}, \text{RemoteFile}) \rightarrow \{\text{ok, Bin} \} | \{\text{error, Reason} \} \]

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

\texttt{recv\_chunk\_start(Pid, RemoteFile) -> ok | \{error, Reason\}}

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

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

\texttt{recv\_chunk(Pid) -> ok | \{ok, Bin\} | \{error, Reason\}}

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

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

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

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

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

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

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

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

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

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

Transfers the file \texttt{LocalFile} to the remote server. If \texttt{RemoteFile} is specified, the name of the remote file is set to \texttt{RemoteFile}; otherwise the name is set to \texttt{LocalFile}.
Inets Reference Manual

Inets Reference Manual

ftp

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 neccessary 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 transfer type is set to binary.

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 function is intended to give an application access to FTP commands that are server specific or that may not be provided by this FTP client.

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

**ERRORS**

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

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

- eclosed  
  The session has been closed.

- econn  
  Connection to remote server prematurely closed.

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

- elogin  
  User not logged in.

- 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].
efnomen File name not allowed [553].

SEE ALSO

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

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

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

### COMMON DATA TYPES

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

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

### HTTP DATA TYPES

Type definitions that are related to HTTP:

For more information about HTTP see rfc 2616

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

SSL DATA TYPES

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

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

Exports

cancel_request(RequestId) -> ok
Types:
  • RequestId = request_id() - A unique identifier as returned by request/4
    Cancels an asynchronous HTTP-request.

request(Url) -> \{ok, Result\} | \{error, Reason\}
Types:
  • Url = url()
  • Result = \{status_line(), headers(), body()\} | \{status_line(), body()\} | request_id()
  • Reason = term()
    Equivalent to http:request(get, \{Url, []\}, [], []).

request(Method, Request, HTTPOptions, Options) -> \{ok, Result\} | \{error, Reason\}
Types:
  • Method = method()
  • Request = request()
  • HTTPOptions - [HttpOption]
  • HttpOption - \{timeout, integer()\} | \{ssl, ssl_options()\} | \{autoredirect, boolean()\}
    autoredirect - is true by default i.e. the client will automatically retrieve the
    information from the new URI and return that as the result instead of a 30X-result
    code. Note that for some 30X-result codes automatic redirect is not allowed in these
    cases the 30X-result will always be returned.
  • Options - \{option()\}
    Option - \{sync, boolean()\} | \{body_format, body_format()\} | \{full_result, boolean()\}
    The request function will be synchronous and return a full http response by default.
body\textunderscore format() = string() \mid binary()

The body\textunderscore format option is only valid for the synchronous request and the default is string. When making an asynchronous request the body will always be received as a binary.

\texttt{Result} = \{\texttt{status\_line(), headers(), body()}\} \mid \{\texttt{status\_code(), body()}\} \mid \texttt{request\_id()}
\texttt{Reason} = \texttt{term()}

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

\texttt{set\_options(Options) -> ok}

Types:

\begin{itemize}
  \item \texttt{Options} = [\texttt{Option}]
  \item \texttt{Option} = \{proxy, \{Proxy, NoProxy\}\} \mid \{max\_sessions, \texttt{MaxSessions}\} \mid \{max\_pipeline\_length, \texttt{MaxPipeline}\} \mid \{pipeline\_timeout, \texttt{PipelineTimeout}\} \mid \{cookies \mid \texttt{CookieMode}\} \mid \{ipv6, \texttt{Ipv6Mode}\}
  \item \texttt{Proxy} = \{Hostname, Port\}
  \item \texttt{Hostname} = \texttt{string()}
    \begin{itemize}
      \item ex: "localhost" or "foo.bar.se"
    \end{itemize}
  \item \texttt{Port} = \texttt{integer()}
    \begin{itemize}
      \item ex: 8080
    \end{itemize}
  \item \texttt{NoProxy} = \{NoProxyDesc\}
    \begin{itemize}
      \item \texttt{NoProxyDesc} = DomainDesc \mid HostName \mid IPDesc
    \end{itemize}
  \item \texttt{DomainDesc} = "*.Domain"
    \begin{itemize}
      \item ex: "*.ericsson.se"
    \end{itemize}
  \item \texttt{IPDesc} = \texttt{string()}
    \begin{itemize}
      \item 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).
    \end{itemize}
  \item \texttt{MaxSessions} = \texttt{integer()}
    \begin{itemize}
      \item Maximum number of persistent connections to a host. Default is 2.
    \end{itemize}
  \item \texttt{MaxPipeline} = \texttt{integer()}
    \begin{itemize}
      \item Maximum number of outstanding requests on the same connection to a host. Default is 2.
    \end{itemize}
  \item \texttt{PipelineTimeout} = \texttt{integer()}
    \begin{itemize}
      \item If a persistent connection is idle longer than the pipeline\_timeout the client will close the connection. Default is 0. The server may also have a such a time out but you should not count on it!
    \end{itemize}
  \item \texttt{CookieMode} = enabled \mid disabled \mid verify
    \begin{itemize}
      \item 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\textunderscore verify\_cookie/2 has to be called for the cookie to be saved.
    \end{itemize}
  \item \texttt{ipv6\_Mode} = enabled \mid disabled
    \begin{itemize}
      \item By default enabled. This should normally be what you want. When it is enabled you can use both ipv4 and ipv6. The option is here to provide a workaround for buggy ipv6 stacks to ensure that ipv4 will always work.
    \end{itemize}
\end{itemize}
Sets options to be used for subsequent requests. Later implementations may support user profiles, but currently these are global settings for all clients running on the same erlang node.

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

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

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

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

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

Types:
- `Url = url()`

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

**SEE ALSO**

RFC 2616, [ssl(3)]
http_base_64

Erlang Module

Implements base 64 encode and decode, see RFC 2045.

COMMON DATA TYPES

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

- \texttt{string()} = list of ASCII characters

Exports

\begin{verbatim}
encode(PlainASCII) -> Base64

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

Encodes a plain ASCII string into base64.

decode(Base64) -> PlainASCII

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

Decodes an base64 encoded string to plain ASCII.
\end{verbatim}
httpd
Erlang Module

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

DATA TYPES

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

The fields of the mod record has the following meaning:

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

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

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

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

method  Type \("\text{GET}\" | \"\text{POST}\" | \"\text{HEAD}\" | \"\text{TRACE}\", that is the HTTP metod.

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

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

http_version  The HTTP version of the request, that is "HTTP/0.9", "HTTP/1.0", or "HTTP/1.1".
The Request-Line as defined in RFC 1945, for example "GET /cgi-bin/find.pl?person=jocke HTTP/1.0".

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

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 Webserver API Callback Functions

Exports

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

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

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

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

Body is the body of the http-response that will be sent back to the client an appropriate header will be appended to the message. StatusCode will be the status code of the response see RFC 2616 for the appropriate values.

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

If Body2 is returned and equal to \{Fun, Arg\} The Web server will try apply/2. on Fun with Arg as argument and expect that the fun either returns a list (Body) that is a HTTP-reponse 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.

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

Types:
- Line = string()
- Directive = \{DirectiveKey, DirectiveValue\}
- DirectiveKey = DirectiveValue = term()
- Reason = term()

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

\begin{verbatim}

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

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

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

Types:
- ConfigDB = ets:table()
- Reason = term()

When httpd shutdown it will try to execute remove/1 in each ewsapi module. The ewsapi programmer may use this to close ets tables, save data, or close down background processes.
\end{verbatim}
Erlang Webserver API Help Functions

Exports

\[
parse\_query(\text{QueryString}) \rightarrow \text{ServerRet}
\]

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

parse\_query/1 parses incoming data to erl and eval scripts (See modules(3) [page 77]) as defined in the standard URL format, that is '+' becomes 'space' and decoding of hexadecimal characters ('%xx').

Erlang Webserver Test and Admin Functions

Exports

\[
\text{start}() \\
\text{start}((\text{Config})) \rightarrow \text{ServerRet} \\
\text{start\_link}() \\
\text{start\_link}((\text{Config})) \rightarrow \text{ServerRet}
\]

Types:
- Config = string() | HttpdOptions
- ServerRet = {ok,Pid} | {ignore} | {error,EReason} | {stop,SReason}
- Pid = pid()
- EReason = {already\_started, Pid} | term()
- SReason = string()
- HttpdOptions = {file,string()} | {debug,Debug}? | {accept\_timeout,integer()}?
- Debug = disable | [DebugOptions]
- DebugOptions = {all\_functions,Modules} | {exported\_functions,Modules} |
  {disable,Modules}
- Modules = [atom()]

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

start/0 and start/0 starts a server as specified in a hard-wired config file, that is start("/var/tmp/server\_root/conf/8888.conf"). Before utilizing start/0 or start\_link/0, copy the example server root\(^1\) to a specific installation directory\(^2\) and you have a server running in no time.

If you copy the example server root to the specific installation directory it is furthermore easy to start an SSL enabled server, that is start("/var/tmp/server\_root/conf/ssl.conf").

\(^1\)In Windows: %INETS_ROOT%\examples\server\_root\ In UNIX: $INETS_ROOT/examples/server\_root/.
\(^2\)In Windows: X:\var\tmp\ In UNIX: /var/tmp/.
restart()
restart(Port) -> ok | {error,Reason}
restart(ConfigFile) -> ok | {error,Reason}
restart(Address,Port) -> ok | {error,Reason}

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

restart restarts the server and reloads its config file.

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

**Note:**
Before the restart function can be called the server must be blocked [page 60].
After restart has been called, the server must be unblocked [page 60].

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

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

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

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

Types:
- Port = integer()
This function is used to block a server. The blocking can be done in two ways, disturbing or non-disturbing.

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

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

Default mode is disturbing.
Default port is 8888

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

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

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

httpd(3) [page 55]
httpd_socket

Erlang Module

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

Exports

deliver(SocketType, Socket, Data) -> Result

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

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

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

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

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

resolve() -> HostName

Types:
- HostName = string()

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

SEE ALSO

httpd(3) [page 55]
httpd_util

Erlang Module

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

Exports

convert_request_date(DateString) -> ErlDate|bad_date

Types:
- DateString = string()
- ErlDate = [{Year,Month,Date},{Hour,Min,Sec}]
- Year = Month = Date = Hour = Min = Sec = integer()
convert_request_date/1 converts DateString to the Erlang date format. DateString must be in one of the three date formats that is defined in the RFC 2616.

create_etag(FileInfo) -> Etag

Types:
- FileInfo = file_info()
- Etag = string()
create_etag/1 calculates the Etag for a file, from it's size and time for last modification. fileinfo is a record defined in kernel/include/file.hrl

decode_base64(Base64String) -> ASCIIString

Types:
- Base64String = ASCIIString = string()

Deprecated use http_base64:decode/1 [page 54]

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 (\(\text{NthDayOfWeek}\)) as an integer (1-7) to an abbreviated string, that is:

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

encode\_base64(ASCIIString) \(\rightarrow\) Base64String

Types:
- ASCIIString = \(\text{string()}\)
- Base64String = \(\text{string()}\)

Deprecated use http\_base\_64:decode/1 [page 54]

flatlength(NestedList) \(\rightarrow\) Size

Types:
- NestedList = \(\text{list()}\)
- Size = \(\text{integer()}\)

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

header(StatusCode, PersistentConn)
header(StatusCode, Date)
header(StatusCode, MimeType, Date)
header(StatusCode, MimeType, PersistentConn, Date) \(\rightarrow\) HTTPHeader

Types:
- StatusCode = \(\text{integer()}\)
- Date = rfc1123\_date()
- MimeType = \(\text{string()}\)
- PersistentConn = true | false

header returns a HTTP 1.1 header string. The StatusCode is one of the status codes defined in RFC 2616 and the Date string is RFC 1123 compliant. (See rfc1123\_date/0 [page 68]).

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

hexlist\_to\_integer(HexString) \(\rightarrow\) Number

Types:
- Number = \(\text{integer()}\)
- HexString = \(\text{string()}\)

hexlist\_to\_integer Convert the Hexadecimal value of HexString to an integer.

integer\_to\_hexlist(Number) \(\rightarrow\) HexString

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

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

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

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

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

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

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

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

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

lookup_mime returns the mime type associated with a specific file suffix as specified in the mime.types file (located in the config directory3).

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

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

3In Windows: %SERVER_ROOT%\conf\mime.types. In UNIX: $SERVER_ROOT/conf/mime.types.
lookup_mime_default returns the mime type associated with a specific file suffix as specified in the mime.types file (located in the config directory\(^4\)). If no appropriate association can be found the value of DefaultType is returned.

message(StatusCode,PhraseArgs,ConfigDB) \rightarrow 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) \rightarrow 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) \rightarrow 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) \rightarrow Description

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

\(^4\)In Windows: %SERVER_ROOT%\conf\mime.types. In UNIX: $SERVER_ROOT/conf/mime.types.
reason_phrase returns the Description of an HTTP 1.1 StatusCode, for example 200 is “OK” and 201 is “Created”. Read RFC 2616 for further information.

rfc1123_date() -> RFC1123Date
rfc1123_date([YYYY,MM,DD],[Hour,Min,Sec])) -> RFC1123Date
Types:
- YYYY = MM = DD = Hour = Min =Sec = integer()
- RFC1123Date = string()
rfc1123_date/0 returns the current date in RFC 1123 format. rfc_date/1 converts the date in the Erlang format to the RFC 1123 date format.

split(String,RegExp,N) -> SplitRes
Types:
- String = RegExp = string()
- SplitRes = {ok, FieldList} | {error, errordesc()}
- Fieldlist = [string()]
- N = integer
split/3 splits the String in N chunks using the RegExp. split/3 is is equivalent to regexp:split/2 with one exception, that is N defines the number of maximum number of fields in the FieldList.

split_script_path(RequestLine) -> Splitted
Types:
- RequestLine = string()
- Splitted = not_a_script | {Path, PathInfo, QueryString}
- Path = QueryString = PathInfo = string()
split_script_path/1 is equivalent to split_path/1 with one exception. If the longest possible path is not a regular, accessible and executable file not_a_script is returned.

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

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

\[\text{suffix}(\text{FileName}) \rightarrow \text{Suffix}\]

**Types:**
- FileName = Suffix = string()

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

\[\text{to\_lower}(\text{String}) \rightarrow \text{ConvertedString}\]

**Types:**
- String = ConvertedString = string()

\text{to\_lower/1} converts upper-case letters to lower-case.

\[\text{to\_upper}(\text{String}) \rightarrow \text{ConvertedString}\]

**Types:**
- String = ConvertedString = string()

\text{to\_upper/1} converts lower-case letters to upper-case.

**SEE ALSO**

httpd(3) [page 55]
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 User Guide [page 4].

path(PathData, ConfigDB, RequestURI) -> Path

Types:
  • PathData = interaction_data()
  • ConfigDB = config_db()
  • RequestURI = Path = string()

path/3 returns the actual file Path in the RequestURI (See RFC 1945). If the interaction data {real_name, {Path, AfterPath}} has been exported by mod_alias, Path is returned. If no interaction data has been exported, ServerRoot is used to generate a file Path. config_db() and interaction_data() are as defined in Inets User Guide [page 4].

real_name(ConfigDB, RequestURI, Aliases) -> Ret

Types:
  • ConfigDB = config_db()
  • RequestURI = string()
  • Aliases = [{FakeName, RealName}]
  • Ret = {ShortPath, Path, AfterPath}
  • ShortPath = Path = AfterPath = string()
real_name/3 traversesAliases, 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_script_path/1 [page 68]. Path is generated from ShortPath, that is the result from default_index/2 [page 70] with ShortPath as an argument.

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

real_script_name(ConfigDB,RequestURI,ScriptAliases) -> Ret

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

real_name/3 traverses ScriptAliases, typically extracted from ConfigDB, and matches each FakeName with RequestURI. If a match is found FakeName is replaced with RealName in the match. If the resulting match is not an executable script not_a_script is returned. If it is a script the resulting script path is in two parts, that is ShortPath and AfterPath as defined in httpd_util:split_script_path/1 [page 68]. config_db() is the server config file in ETS table format as described in Inets User Guide [page 4].
mod_auth

Erlang Module

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

Exports

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

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

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

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

Types:
User Name = 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()
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/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/3, list_group_members/4 and list_group_members/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.
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} | 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} | 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:
update_password/5 and update_password/6 Updates the AuthAccessPassword for the specified directory. If NewPassword is equal to "NoPassword" no password is required to change authorisation data. If NewPassword is equal to "DummyPassword" no changes can be done without changing the password first.

SEE ALSO

httpd(3) [page 55], mod_alias(3) [page 70],
mod_esi

Erlang Module

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

Exports

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

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

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

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

```erlang
Module:Function(Env, Input) -> Response
Types:
- Env = [EnvironmentDirectives] ++ ParsedHeader
- EnvironmentDirectives = [{Key, Value}]
- Key = query_string | content_length | server_software | gateway_interface | server_protocol | server_port | request_method | remote_addr | script_name.
- Input = 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 kept for backwards compability. For new development Module:Function/3 should be used.
mod_security

Erlang Module

Security Audit and Trailing Functionality

Exports

\[
\text{list\_auth\_users}(\text{Port}) \rightarrow \text{Users} | []
\]
\[
\text{list\_auth\_users}(\text{Address, Port}) \rightarrow \text{Users} | []
\]
\[
\text{list\_auth\_users}(\text{Port, Dir}) \rightarrow \text{Users} | []
\]
\[
\text{list\_auth\_users}(\text{Address, Port, Dir}) \rightarrow \text{Users} | []
\]

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

\text{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.

\[
\text{list\_blocked\_users}(\text{Port}) \rightarrow \text{Users} | []
\]
\[
\text{list\_blocked\_users}(\text{Address, Port}) \rightarrow \text{Users} | []
\]
\[
\text{list\_blocked\_users}(\text{Port, Dir}) \rightarrow \text{Users} | []
\]
\[
\text{list\_blocked\_users}(\text{Address, Port, Dir}) \rightarrow \text{Users} | []
\]

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

\text{list\_blocked\_users/1, list\_blocked\_users/2 and list\_blocked\_users/3} returns a list of users that are currently blocked from access.

\[
\text{block\_user}(\text{User, Port, Dir, Seconds}) \rightarrow \text{true} | \{\text{error, Reason}\}
\]
\[
\text{block\_user}(\text{User, Address, Port, Dir, Seconds}) \rightarrow \text{true} | \{\text{error, Reason}\}
\]

Types:
- \text{User} = \text{string()}
- \text{Port} = \text{integer()}
- \text{Address} = \{A,B,C,D\} | \text{string()} | \text{undefined}
block_user/4 and block_user/5 blocks the user User from the directory Dir for a specified amount of time.

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

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

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

The SecurityCallbackModule

The SecurityCallbackModule is a user written module that can receive events from the mod_security Erlang Webserver API module. This module only exports one function, event/4 [page 80], 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() | undefined
- 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.
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.

**DATA TYPES**

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

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

- `{debug, Level} Level = none | brief | normal | verbose | all`
  Controls the level of debug printouts. The default is `none`.
- `{host, Host} Host = hostname() see [inet(3)]`
  The name or IP address of the host where the TFTP daemon resides. This option is only used by the client.
- `{port, Port} Port = int()`
  The TFTP port where the daemon listens. It defaults to the standardized number 69. On the server side it may sometimes make sense to set it to 0, which means that the daemon just will pick a free port (which one is returned by the `info/1` function).
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.

Options = [Opt]

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.

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

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.

MaxTsize = int() | infinity

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

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 = string()
TftpVal = string()

The name and value of a TFTP option.

Feature = Mode | TftpKey

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.

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 during the transfer. See read_file/3 [page 84] and write_file/3 [page 84].

The callback module must implement the tftp behavior, CALLBACK FUNCTIONS [page 85].

Exports

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

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

Starts a daemon process which listens for udp packets on a port. When it receives a request for read or write it spawns a temporary server process which handles the actual transfer of the (virtual) file.
read_file(RemoteFilename, LocalFilename, Options) -> \{ok, LastCallbackState\} | \{error, Reason\}

Types:
- RemoteFilename = string()
- LocalFilename = binary \| string()
- Options = \{option()\}
- LastCallbackState = term()
- Reason = term()

Reads a (virtual) file RemoteFilename from a TFTP server.
If LocalFilename is the atom binary, tftp_binary is used as callback module. It concatenates all transferred blocks and returns them as one single binary in LastCallbackState.
If LocalFilename is a string and there are no registered callback modules, tftp_file is used as callback module. It writes each transferred block to the file named LocalFilename and returns the number of transferred bytes in LastCallbackState.
If LocalFilename is a string and there are registered callback modules, LocalFilename is tested against the regexps of these and the callback module corresponding to the first match is used, or an error tuple is returned if no matching regexp is found.

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

Types:
- RemoteFilename = string()
- LocalFilename = binary() \| string()
- Options = \{option()\}
- LastCallbackState = term()
- Reason = term()

Writes a (virtual) file RemoteFilename to a TFTP server.
If LocalFilename is a binary, tftp_binary is used as callback module. The binary is transferred block by block and the number of transferred bytes is returned in LastCallbackState.
If LocalFilename is a string and there are no registered callback modules, tftp_file is used as callback module. It reads the file named LocalFilename block by block and returns the number of transferred bytes in LastCallbackState.
If LocalFilename is a string and there are registered callback modules, LocalFilename is tested against the regexps of these and the callback module corresponding to the first match is used, or an error tuple is returned if no matching regexp is found.

info(Pid) -> undefined \| Options

Types:
- Options = \{option()\}

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

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

Types:
- Reason = term()
  Starts the Inets application.

CALLBACK FUNCTIONS

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

On the server side the callback interaction starts with a call to \texttt{open/5} with the registered initial callback state. \texttt{open/5} is expected to open the (virtual) file. Then either the \texttt{read/1} or \texttt{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 \texttt{read/1} or \texttt{write/2} functions is expected to close the (virtual) file and return its last state. The \texttt{abort/3} function is only used in error situations. \texttt{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 \texttt{prepare/5} with the same arguments as \texttt{open/5} takes. \texttt{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 \texttt{open/5} function on the client side. On the client side the \texttt{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 \texttt{read/1} or \texttt{write/2} the returned state is forwarded to the user and returned from \texttt{read_file/3} or \texttt{write_file/3}.

Exports

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

Types:
- \texttt{Access = read | write}
- \texttt{Filename = string()}
- \texttt{Mode = string()}
- \texttt{SuggestedOptions = AcceptedOptions = [{Key, Value}]}  
- \texttt{Key = Value = string()}
- \texttt{InitialState = [ \{root_dir, string()\} ]}
- \texttt{NewState = term()}
- \texttt{Code = undef | enoent | eacces | enospc}
- \texttt{| badop | eexist | baduser | badopt}
- \texttt{| int()}
- \texttt{Text = string()}

Inets
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.

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

Types:
- 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.

```prolog
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.

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

Types:
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.

\begin{verbatim}
abort(Code, Text, State) -> ok
\end{verbatim}

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

Invoked when the file transfer is aborted.

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

HTTP
Hypertext Transfer Protocol.

RFC
A “Request for Comments” used as a proposed standard by IETF.
Index of Modules and Functions

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

abort/3
  ftp, 87
account/2
  ftp, 41
add_group_member/3
  mod_auth, 73
add_group_member/4
  mod_auth, 73
add_group_member/5
  mod_auth, 73
add_user/2
  mod_auth, 72
add_user/5
  mod_auth, 72
add_user/6
  mod_auth, 72
append/3
  ftp, 41
append_bin/3
  ftp, 41
append_chunk/2
  ftp, 41
append_chunk_end/1
  ftp, 42
append_chunk_start/2
  ftp, 41
block/0
  httpd, 59
block/1
  httpd, 59
block/2
  httpd, 59
block/3
  httpd, 59
block/4
  httpd, 59
block_user/4
  mod_security, 79
block_user/5
  mod_security, 79
cancel_request/1
  http, 51
cd/2
  ftp, 42
check_enum/2
  httpd_conf, 61
clean/1
  httpd_conf, 61
close/1
  ftp, 42
convert_request_date/1
  httpd_util, 64
cookie_header/1
  http, 53
create_etag/1
  httpd_util, 64
custom_clean/3
  httpd_conf, 61
day/1
  httpd_util, 64
decode/1
  http_base64, 54
decode_base64/1
  httpd_util, 64
decode_hex/1
Index of Modules and Functions

httpd_util, 64
default_index/2
  mod_alias, 70
delete/2
  ftp, 42
delete_group/2
  mod_auth, 75
delete_group/4
  mod_auth, 75
delete_group_member/3
  mod_auth, 74
delete_group_member/4
  mod_auth, 74
delete_group_member/5
  mod_auth, 74
delete_user/2
  mod_auth, 72
delete_user/3
  mod_auth, 72
delete_user/4
  mod_auth, 72
deliver/2
  mod_esi, 77
deliver/3
  httpd_socket, 63
encode/1
  http_base64, 54
encode_base64/1
  httpd_util, 65
event/4
  mod_security, 80
event/5
  mod_security, 80
flatlength/1
  httpd_util, 65
formatter/1
  ftp, 42
ftp
  account/2, 41
  append/3, 41
  append_bin/3, 41
  append_chunk/2, 41
  append_chunk_end/1, 42
  append_chunk_start/2, 41
cd/2, 42
close/1, 42
delete/2, 42
formatter/1, 42
lcd/2, 42
lpwd/1, 42
ls/2, 43
mkdir/2, 43
nlist/2, 43
open/2, 43
open/3, 43
pwd/1, 45
quote/2, 48
recv/3, 45
recv_bin/2, 45
recv_chunk/1, 46
recv_chunk_start/2, 46
rename/3, 46
rmdir/2, 46
send/3, 46
send_bin/3, 47
send_chunk/2, 47
send_chunk_end/1, 47
send_chunk_start/2, 47
type/2, 47
user/3, 47
user/4, 48
get_user/2
  mod_auth, 73
get_user/3
  mod_auth, 73
get_user/4
  mod_auth, 73
header/2
  httpd_util, 65
header/3
  httpd_util, 65
header/4
  httpd_util, 65
hexlist_to_integer/1
  httpd_util, 65
http
  cancel_request/1, 51
  cookie_header/1, 53
  request/1, 51
  request/4, 51

Inets
set_options/1, 52
verify_cookie/2, 53

httpd
    block/0, 59
    block/1, 59
    block/2, 59
    block/3, 59
    block/4, 59
    Module:do/1, 56
    Module:load/2, 57
    Module:remove/1, 57
    Module:store/3, 57
    parse_query/1, 58
    restart/0, 59
    restart/1, 59
    restart/2, 59
    start/0, 58
    start/1, 58
    start_link/0, 58
    start_link/1, 58
    stop/0, 59
    stop/1, 59
    stop/2, 59
    unblock/0, 60
    unblock/1, 60
    unblock/2, 60

httpd_conf
    check_enum/2, 61
    clean/1, 61
    custom_clean/3, 61
    is_directory/1, 61
    is_file/1, 62
    make_integer/1, 62

httpd_socket
    deliver/3, 63
    peername/2, 63
    resolve/0, 63

httpd_util
    convert_request_date/1, 64
    create_etag/1, 64
    day/1, 64
    decode_base64/1, 64
    decode_hex/1, 64
    encode_base64/1, 65
    flatlength/1, 65
    header/2, 65
    header/3, 65
    header/4, 65
    hexlist_to_integer/1, 65
    integer_tohexlist/1, 65
    key1search/2, 66
    key1search/3, 66
    lookup/2, 66
    lookup/3, 66
    lookup_mime/2, 66
    lookup_mime/3, 66
    lookup_mime_default/2, 66
    lookup_mime_default/3, 66
    message/3, 67
    month/1, 67
    multi_lookup/2, 67
    reason_phrase/1, 67
    rfc1123_date/0, 68
    rfc1123_date/6, 68
    split/3, 68
    split_path/1, 68
    split_script_path/1, 68
    strip/1, 68
    suffix/1, 69
    to_lower/1, 69
    to_upper/1, 69

info/1
    tftp, 84
    integer_tohexlist/1
        httpd_util, 65
is_directory/1
    httpd_conf, 61
is_file/1
    httpd_conf, 62
key1search/2
    httpd_util, 66
key1search/3
    httpd_util, 66

lcd/2
    ftp, 42
    list_auth_users/1
        mod_security, 79
    list_auth_users/2
        mod_security, 79
    list_auth_users/3
        mod_security, 79
    list_blocked_users/1

Inets 93
Index of Modules and Functions

`mod_security`, 79

list_blocked_users/2
  `mod_security`, 79

list_blocked_users/3
  `mod_security`, 79

list_group_members/2
  `mod_auth`, 74

list_group_members/3
  `mod_auth`, 74

list_group_members/4
  `mod_auth`, 74

list_groups/1
  `mod_auth`, 75

list_groups/2
  `mod_auth`, 75

list_groups/3
  `mod_auth`, 75

list_users/1
  `mod_auth`, 73

list_users/3
  `mod_auth`, 73

lookup/2
  `httpd_util`, 66

lookup/3
  `httpd_util`, 66

lookup_mime/2
  `httpd_util`, 66

lookup_mime/3
  `httpd_util`, 66

lookup_mime_default/2
  `httpd_util`, 66

lookup_mime_default/3
  `httpd_util`, 66

`lpwd`/1
  `ftp`, 42

`ls`/2
  `ftp`, 43

make_integer/1
  `httpd_conf`, 62

message/3
  `httpd_util`, 67

mkdir/2

`ftp`, 43

`mod_alias`
  default_index/2, 70
  path/3, 70
  real_name/3, 70
  real_script_name/3, 71

`mod_auth`
  add_group_member/3, 73
  add_group_member/4, 73
  add_group_member/5, 73
  add_user/2, 72
  add_user/5, 72
  add_user/6, 72
  delete_group/2, 75
  delete_group/4, 75
  delete_group_member/3, 74
  delete_group_member/4, 74
  delete_group_member/5, 74
  delete_user/2, 72
  delete_user/3, 72
  delete_user/4, 72
  get_user/2, 73
  get_user/3, 73
  get_user/4, 73

list_group_members/2, 74
list_group_members/3, 74
list_group_members/4, 74

list_users/1
  `mod_auth`, 73

list_users/2
  `mod_auth`, 73

list_users/3
  `mod_auth`, 73

update_password/5, 75
update_password/6, 75

`mod_esi`
  deliver/2, 77
  Module:Function/2, 78
  Module:Function/3, 77

`mod_security`
  block_user/4, 79
  block_user/5, 79
  event/4, 80
  event/5, 80
  list_auth_users/1, 79
  list_auth_users/2, 79
  list_auth_users/3, 79
  list_blocked_users/1, 79
  list_blocked_users/2, 79
  list_blocked_users/3, 79
  unblock_user/2, 80
  unblock_user/3, 80
Index of Modules and Functions

unblock_user/4, 80
Module:do/1
    httpd , 56
Module:Function/2
    mod_esi , 78
Module:Function/3
    mod_esi , 77
Module:load/2
    httpd , 57
Module:remove/1
    httpd , 57
Module:store/3
    httpd , 57
month/1
    httpd util , 67
multi_lookup/2
    httpd util , 67
nlist/2
    ftp , 43
open/2
    ftp , 43
open/3
    ftp , 43
open/5
    tftp , 86
parse_query/1
    httpd , 58
path/3
    mod_alias , 70
peername/2
    httpd_socket , 63
prepare/5
    tftp , 85
pwd/1
    ftp , 45
quote/2
    ftp , 48
read/1
    tftp , 86
read_file/3
    tftp , 84
real_name/3
    mod_alias , 70
real_script_name/3
    mod_alias , 71
reason_phrase/1
    httpd util , 67
recv/3
    ftp , 45
recv_bin/2
    ftp , 45
recv_chunk/1
    ftp , 46
recv_chunk_start/2
    ftp , 46
rename/3
    ftp , 46
request/1
    http , 51
request/4
    http , 51
resolve/0
    httpd_socket , 63
restart/0
    httpd , 59
restart/1
    httpd , 59
restart/2
    httpd , 59
rfc1123_date/0
    httpd util , 68
rfc1123_date/6
    httpd util , 68
rmdir/2
    ftp , 46
send/3
    ftp , 46
send_bin/3
    ftp , 47
send_chunk/2
    ftp , 47
send_chunk_end/1
    ftp , 47
ftp , 47
send_chunk_start/2
ftp , 47
set_options/1
http , 52
split/3
httpd_util , 68
split_path/1
httpd_util , 68
split_script_path/1
httpd_util , 68
start/0
httpd , 58
tftp , 84
start/1
httpd , 58
tftp , 83
start_link/0
httpd , 58
start_link/1
httpd , 58
stop/0
httpd , 59
stop/1
httpd , 59
stop/2
httpd , 59
strip/1
httpd_util , 68
suffix/1
httpd_util , 69
tftp
abort/3, 87
info/1, 84
open/5, 86
prepare/5, 85
read/1, 86
read_file/3, 84
start/0, 84
start/1, 83
write/2, 86
write_file/3, 84
to_upper/1
httpd_util , 69
type/2
ftp , 47
unblock/0
httpd , 60
unblock/1
httpd , 60
unblock/2
httpd , 60
unblock_user/2
mod_security , 80
unblock_user/3
mod_security , 80
unblock_user/4
mod_security , 80
update_password/5
mod_auth , 75
update_password/6
mod_auth , 75
user/3
ftp , 47
user/4
ftp , 48
verify_cookie/2
http , 53
write/2
tftp , 86
write_file/3
tftp , 84