1 Inets User's Guide

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

- An HTTP client and server

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

1.1 Introduction

1.1.1 Purpose

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

- An HTTP client and server

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

1.1.2 Prerequisites

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

1.2 Inets

1.2.1 Service Concept

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

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

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

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

1.3 HTTP Client

1.3.1 Configuration

The HTTP client default profile is started when the Inets application is started and is then available to all processes on that Erlang node. Other profiles can also be started at application startup, or profiles can be started and stopped dynamically in runtime. Each client profile spawns a new process to handle each request, unless a persistent connection...
1.3 HTTP Client

can be used with or without pipelining. The client adds a host header and an empty te header if there are no such headers present in the request.

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

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

```
{inets, [{services, [[httpc, PropertyList]]]}]
```
For valid properties, see httpc(3).

1.3.2 Getting Started

Start Inets:

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

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

Example:

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

The following is an ordinary synchronous request:

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

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

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

The following is a https request and with verification of the host:

```
5 > {ok, {{Version, 200, ReasonPhrase}, Headers, Body}} =
   httpc:request(get, {"https://www.erlang.org", []}, [{ssl, httpc:ssl_verify_host_options(true)}], []).
```

The following is an ordinary asynchronous request:

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

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

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

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

This sends a request with a specified connection header:

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

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

1.4 HTTP server

1.4.1 Configuration

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

The server implements numerous features, such as:

- Secure Sockets Layer (SSL)
- Erlang Scripting Interface (ESI)
- Common Gateway Interface (CGI)
- User Authentication (using Mnesia, Dets or plain text database)
- Common Logfile Format (with or without disk_log(3) support)
- URL Aliasing
- Action Mappings
- Directory Listings

The configuration of the server is provided as an Erlang property list.

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

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

The following is to be put in the Erlang node application configuration file to start an HTTP server at application startup:
The server is configured using an Erlang property list. For the available properties, see httpd(3).

The available configuration properties are as follows:

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

Here:

{proplist_file, file()}

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

{debug, debug()}

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

{accept_timeout, integer()}

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

### 1.4.2 Getting Started

#### Start Inets:

1 > inets:start().

ok

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

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

{ok, 0.79.0}

Call info:

3 > httpd:info(Pid).

```
[[mime_types, [{"html", "text/html"}, 
               {"htm", "text/html"}]], 
  {server_name, "httpd_test"},
  {bind_address, [127,0,0,1]},
  {server_root, "/tmp"}, 
  {port, 59408},
  {document_root, "/tmp/htdocs"}]
```

Reload the configuration without restarting the server:
1.4 HTTP server

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

Note:

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

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

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

Alternative:

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

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

1.4.3 Dynamic Web Pages

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

**CGI scripts**

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

**ESI-functions**

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

**CGI Version 1.1, RFC 3875**

The module mod_cgi enables execution of **CGI scripts** on the server. A file matching the definition of a ScriptAlias config directive is treated as a CGI script. A CGI script is executed by the server and its output is returned to the client.

The CGI script response comprises a message header and a message body, separated by a blank line. The message header contains one or more header fields. The body can be empty.

Example:

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

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

Support for CGI-1.1 is implemented in accordance with RFC 3875.
1.4 HTTP server

ESI

The Erlang server interface is implemented by module mod_esi.

ERL Scheme

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

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

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

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

1.4.4 Logging

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

```
remotehost rfc931 authuser [date] "request" status bytes
```

Here:

- **remotehost**
  - Remote hostname.
- **rfc931**
  - The client remote username (RFC 931).
- **authuser**
  - The username used for authentication.
- **[date]**
  - Date and time of the request (RFC 1123).
- "**request**"
  - The request line exactly as it came from the client (RFC 1945).
- **status**
  - The HTTP status code returned to the client (RFC 1945).
- **bytes**
  - The content-length of the document transferred.

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

```
[date] access to path failed for remotestart1, reason: reason
```

1.4.5 Erlang Web Server API

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

- Setting up connections, sending and receiving data.
- URI to filename translation.
- Authentication/access checks.
- Retrieving/generating the response.
- Logging.
To provide customization and extensibility of the request handling of the HTTP servers, most of these steps are handled by one or more modules. These modules can be replaced or removed at runtime and new ones can be added. For each request, all modules are traversed in the order specified by the module directive in the server configuration file. Some parts, mainly the communication-related steps, are considered server core functionality and are not implemented using the Erlang web server API. A description of functionality implemented by the Erlang webservice API is described in Section Inets Web Server Modules.

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

**Note:**

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

### API Description

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

- `do/1` (mandatory) - the function called when a request is to be handled
- `load/2`
- `store/2`
- `remove/1`

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

### 1.4.6 Inets Web Server Modules

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

**mod_action - Filetype/Method-Based Script Execution**

This module runs CGI scripts whenever a file of a certain type or HTTP method (see [RFC 1945](https://tools.ietf.org/html/rfc1945)) is requested.

Uses the following Erlang Web Server API interaction data:

- `real_name` - from `mod_alias`.

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

```erlang
{new_request_uri, RequestURI}
```

An alternative `RequestURI` has been generated.

**mod_alias - URL Aliasing**

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

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

```erlang
{real_name, PathData}
```

`PathData` is the argument used for API function `mod_alias:path/3`.
mod_auth - User Authentication

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

Uses the following Erlang Web Server API interaction data:

- `real_name` - from mod_alias

Exports the following Erlang Web Server API interaction data:

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

Mnesia As Authentication Database

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

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

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

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

To create the Mnesia tables, we use two records defined in `mod_auth.hrl`, so that file must be included. `first_start/0` creates a schema that specifies on which nodes the database is to reside. Then it starts Mnesia and creates the tables. The first argument is the name of the tables, the second argument is a list of options of how to create the table, see `mnesia(3)`, documentation for more information. As the implementation of the `mod_auth_mnesia` saves one row for each user, the type must be `bag`. When the schema and the tables are created, function `mnesia:start/0` is used to start Mnesia and waits for the tables to be loaded. Mnesia uses the directory specified as `mnesia_dir` at startup if specified, otherwise Mnesia uses the current directory. For security reasons, ensure that the Mnesia tables are stored outside the document tree of the HTTP server. If they are placed in the directory which it protects, clients can download the tables. Only the Dets and Mnesia storage methods allow writing of dynamic user data to disk. `plain` is a read only method.

mod_cgi - CGI Scripts

This module handles invoking of CGI scripts.
mod_dir - Directories
This module generates an HTML directory listing (Apache-style) if a client sends a request for a directory instead of a file. This module must be removed from the Modules config directive if directory listings is unwanted.

Uses the following Erlang Web Server API interaction data:
• real_name - from mod_alias

Exports the following Erlang Web Server API interaction data:
{mime_type, MimeType}
The file suffix of the incoming URL mapped into a MimeType.

mod_disk_log - Logging Using Disk_Log.
Standard logging using the "Common Logfile Format" and disk_log(3).

Uses the following Erlang Web Server API interaction data:
• remote_user - from mod_auth

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

Uses the following Erlang web server API interaction data:
• remote_user - from mod_auth

Exports the following Erlang web server API interaction data:
{mime_type, MimeType}
The file suffix of the incoming URL mapped into a MimeType

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

Uses the following Erlang web server API interaction data:
• real_name - from mod_alias

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

Uses the following Erlang Web Server API interaction data:
• real_name - from mod_alias

mod_log - Logging Using Text Files.
Standard logging using the "Common Logfile Format" and text files.

Uses the following Erlang Web Server API interaction data:
• remote_user - from mod_auth

mod_range - Requests with Range Headers
This module responds to requests for one or many ranges of a file. This is especially useful when downloading large files, as a broken download can be resumed.
Notice that request for multiple parts of a document report a size of zero to the log file.

Uses the following Erlang Web Server API interaction data:

• real_name - from mod_alias

**mod_response_control - Requests with If* Headers**

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

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

- If-modified
- If-Unmodified
- If-Match
- If-Nomatch

Uses the following Erlang Web Server API interaction data:

• real_name - from mod_alias

Exports the following Erlang Web Server API interaction data:

{if_range, send_file}

The conditions for the range request are not fulfilled. The response must not be treated as a range request, instead it must be treated as an ordinary get request.

**mod_security - Security Filter**

The mod_security module serves as a filter for authenticated requests handled in mod_auth(3). It provides a possibility to restrict users from access for a specified amount of time if they fail to authenticate several times. It logs failed authentication as well as blocking of users, and it calls a configurable callback module when the events occur.

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

**mod_trace - TRACE Request**

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

Inets is a container for an HTTP client and server.
This module provides the most basic API to the clients and servers that are part of the Inets application, such as start and stop.

DATA TYPES

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

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

Exports

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

Types:

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

Returns a list of currently running services.

**Note:**

Services started as `stand_alone` are not listed.

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

Types:

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

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

```
service_names() -> [Service]
```

Types:

- `Service = service()`

Returns a list of available service names.

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

Types:
Type = permanent | transient | temporary

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

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

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

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

**Note:**

Dynamically started services are not handled by application takeover and failover behavior when Inets is run as a distributed application. Nor are they automatically restarted when the Inets application is restarted. As long as the Inets application is operational, they are supervised and can be soft code upgraded.

A service started as stand_alone, that is, the service is not started as part of the Inets application, lose all OTP application benefits, such as soft upgrade. The stand_alone-service is linked to the process that started it. Usually some supervision functionality is still in place and in some sense the calling process becomes the top supervisor.

stop() -> ok

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

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

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

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

**SEE ALSO**

httpc(3), httpd(3)
This module provides the API to an HTTP/1.1 compatible client according to RFC 2616. Caching is not supported.

**Note:**

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

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

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

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

### HTTP CLIENT SERVICE START/STOP

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

- `{profile, Profile :: atom() | pid()}
  - Name of the profile. This option is mandatory.
- `{data_dir, Path :: string()}
  - Directory where the profile can save persistent data. If omitted, all cookies are treated as session cookies. Path represents a file path or directory path.

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

**Warning:**

Please note that httpc normalizes input URIs before internal processing and special care shall be taken when the URI has percent ("%”) characters. A percent serves as the indicator for percent-encoded octets and it must be percent-encoded as "%25" for that octet to be used as data within the URI.

For example, in order to send an HTTP GET request with the URI http://localhost/foo%25bar, the percent character must be percent-encoded when creating the request: httpc:request("http://localhost/foo%2525bar").

### Exports

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

Types:

  - `RequestId = any()`
  - A unique identifier as returned by request/4
Profile = atom() | pid()
When started stand_alone only the pid can be used.

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

cookie_header(Url) -> HttpHeader | {error, Reason}
cookie_header(Url, ProfileOrOpts) -> HttpHeader | {error, Reason}
Types:
  Url = uri_string:uri_string()
  HttpHeader = {Field :: [byte()], Value :: binary() | iolist()}
  ProfileOrOpt = Profile | Opt
  Profile = atom() | pid()
  When started stand_alone only the pid can be used.
  Opt = [CookieHeaderOpt]
  CookieHeaderOpt = {ipv6_host_with_brackets, boolean()}
  Reason = term()

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

Option ipv6_host_with_brackets deals with how to parse IPv6 addresses. For details, see argument Options of request/[4,5].

cookie_header(Url, Opt, Profile) -> HttpHeader | {error, Reason}
Types:
  Url = uri_string:uri_string()
  HttpHeader = {Field :: [byte()], Value :: binary() | iolist()}
  Profile = atom() | pid()
  When started stand_alone only the pid can be used.
  Opt = [CookieHeaderOpt]
  CookieHeaderOpt = {ipv6_host_with_brackets, boolean()}
  Reason = term()

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

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

get_options(OptionItems) -> {ok, Values} | {error, Reason}
get_options(OptionItems, Profile) ->
  {ok, Values} | {error, Reason}
Types:
OptionItems = all | [OptionItem]
OptionItem =
  proxy | https_proxy | max_sessions | keep_alive_timeout |
  max_keep_alive_length | pipeline_timeout |
  max_pipeline_length | cookies | ipfamily | ip | port |
  socket_opts | verbose | unix_socket
Values = [{OptionItem, term()}]
Profile = atom() | pid()
  When started stand_alone only the pid can be used.
Reason = term()

Retrieves the options currently used by the client.

info() -> list() | {error, Reason}
info(Profile) -> list() | {error, Reason}
Types:
  Reason = term()
  Profile = atom() | pid()
  When started stand_alone only the pid can be used.

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

reset_cookies() -> Void
reset_cookies(Profile) -> Void
Types:
  Profile = atom() | pid()
  When started stand_alone only the pid can be used.
  Void = term()

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

request(Url :: uri_string:uri_string()) ->
  {ok, Result} | {error, term()}
request(Url, Profile) -> {ok, Result} | {error, term()}
Types:
  Url = uri_string:uri_string()
  Profile = atom() | pid()
  When started stand_alone only the pid can be used.
Result =
    {StatusLine, [HttpHeader], HttpBodyResult} |
    {StatusCode, HttpBodyResult} |
    RequestId | saved_to_file
HttpHeader = {Field :: [byte()], Value :: binary() | iolist()}
HttpBodyResult = uri_string:uri_string() | binary()
StatusLine = {HttpVersion, StatusCode, string()}
HttpVersion = uri_string:uri_string()
StatusCode = integer() >= 0
RequestId = any()

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

request(Method, Request, HttpOptions, Options) ->
    {ok, Result} | {error, term()}
request(Method, Request, HttpOptions, Options, Profile) ->
    {ok, Result} | {error, term()}

Types:
Method =  head | get | put | patch | post | trace | options | delete

Request = {uri_string:uri_string(), [HttpHeader]} |
{uri_string:uri_string(),
  [HttpHeader],
  ContentType :: uri_string:uri_string(),
  HttpBody}

HttpBody = iolist() |
binary() | 
{fun((Accumulator :: term()) ->
  eof | {ok, iolist(), Accumulator :: term()}),
Accumulator :: term()} |
{chunkify,
  fun((Accumulator :: term()) ->
    eof | {ok, iolist(), Accumulator :: term()}),
Accumulator :: term()}

HttpHeader = {Field :: [byte()], Value :: binary() | iolist()}

HttpOptions = [HttpOption]

HttpOption = 
{timeout, timeout()} |
{connect_timeout, timeout()} |
{ssl, [ssl:tls_option()]} |
{autoredirect, boolean()} |
{proxy_auth, {string(), string()}} |
{version, HttpVersion} |
{relaxed, boolean()}

Options = [OptionRequest]

OptionRequest = 
{sync, boolean()} |
{stream, StreamTo} |
{body_format, BodyFormat} |
{full_result, boolean()} |
{headers_as_is, boolean()} |
{socket_opts, [SocketOpt]} |
{receiver, Receiver} |
{ipv6_host_with_brackets, boolean()}

StreamTo = none | self | {self, once} | file:name_all()

BodyFormat = string | binary

SocketOpt = term()

Receiver = 
  pid() |
  fun((term()) -> term()) |
  {ReceiverModule :: atom(),
   ReceiverFunction :: atom(),
   ReceiverArgs :: list()}

Profile = atom() | pid()

When Profile is stand_alone only the pid can be used.
HttpVersion = uri_string:uri_string()

Result =
  {StatusLine, [HttpHeader], HttpBodyResult} |
  {StatusCode, HttpBodyResult} |
  RequestId | saved_to_file

StatusLine = {HttpVersion, StatusCode, string()}

StatusCode = integer() >= 0

HttpBodyResult = uri_string:uri_string() | binary()

RequestId = any()

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

When Profile is stand-alone only the pid can be used.

HTTP options:

timeout
  Time-out time for the request.
  The clock starts ticking when the request is sent.
  Time is in milliseconds.
  Default is infinity.

connect_timeout
  Connection time-out time, used during the initial request, when the client is connecting to the server.
  Time is in milliseconds.
  Default is the value of option timeout.

ssl
  This is the SSL/TLS connecting configuration option.
  Default value is obtained by calling httpc:ssl_verify_host_options(true). See ssl:connect/[2,3,4] for available options.

autoredirect
  The client automatically retrieves the information from the new URI and returns that as the result, instead of a 30X-result code.
  For some 30X-result codes, automatic redirect is not allowed. In these cases the 30X-result is always returned.
  Default is true.

proxy_auth
  A proxy-authorization header using a tuple where the first element is the username and the second element of the tuple is the password added to the request.

version
  Can be used to make the client act as an HTTP/1.0 client. By default this is an HTTP/1.1 client. When using HTTP/1.0 persistent connections are not used.
  Default is the string "HTTP/1.1".

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

Default is false.

Options details:

sync

Option for the request to be synchronous or asynchronous.

Default is true.

stream

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

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

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

Default is none.

body_format

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

Default is string.

car_result

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

Default is true.

headers_as_is

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

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

Default is false.

socket_opts

Socket options to be used for this request.

See the options used by gen_tcp(3) and ssl(3)

Overrides any value set by function set_options.

The validity of the options is not checked by the HTTP client they are assumed to be correct and passed on to ssl application and inet driver, which may reject them if they are not correct.
Note:

Persistent connections are not supported when setting the `socket_opts` option. When `socket_opts` is not set the current implementation assumes the requests to the same host, port combination will use the same socket options.

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

**receiver**

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

**pid()**

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

**function/1**

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

{Module, Function, Args}

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

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

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

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

**ipv6_host_with_brackets**

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

Default is `false`.

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

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

**Types:**
Options = [Option]
Option =
  {proxy, {Proxy, NoProxy}} |  
  {https_proxy, {Proxy, NoProxy}} |  
  {max_sessions, MaxSessions} |  
  {max_keep_alive_length, MaxKeepAlive} |  
  {keep_alive_timeout, KeepAliveTimeout} |  
  {max_pipeline_length, MaxPipeline} |  
  {pipeline_timeout, PipelineTimeout} |  
  {cookies, CookieMode} |  
  {ipfamily, IpFamily} |  
  {ip, IpAddress} |  
  {port, Port} |  
  {socket_opts, [SocketOpt]} |  
  {verbose, VerboseMode} |  
  {unix_socket, UnixSocket}
Profile = atom() | pid()
SocketOpt = term()
Proxy = {HostName, Port}
Port = integer() >= 0
NoProxy = [DomainDesc | HostName | IpAddressDesc]
MaxSessions = MaxKeepAlive = KeepAliveTimeout = MaxPipeline = PipelineTimeout = integer()
CookieMode = enabled | disabled | verify
IpFamily = inet | inet6 | local | inet6fb4
IpAddressDesc = uri_string:uri_string()
IpAddress = inet:ip_address()
VerboseMode = false | verbose | debug | trace
UnixSocket = string()
Reason = term()
DomainDesc = string()
HostName = uri_string:uri_string()

Sets options to be used for subsequent requests.

HostName
  Example: "localhost" or "foo.bar.se"
DomainDesc
  Example "*.Domain" or "*.ericsson.se"
IpAddressDesc
  Example: "134.138" or ":[FEDC:BA98=" (all IP addresses starting with 134.138 or FEDC:BA98),
  "66.35.250.150" or "[2010:836B:4179::836B:4179]" (a complete IP address). proxy defaults to
  (undefined, []), that is, no proxy is configured and https_proxy defaults to the value of proxy.
MaxSessions
  MaxSessions Maximum number of persistent connections to a host. Default is 2.
MaxKeepAlive
  MaxKeepAlive Maximum number of outstanding requests on the same connection to a host. Default is 5.
KeepAliveTimeout

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

MaxPipeline

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

PipelineTimeout

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

CookieMode

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

IpFamily

Default is inet. With inet6fb4 option, IPv6 will be preferred but if connection fails, an IPv4 fallback connection attempt will be made.

IpAddress

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

Port

Example: 8080. Local port number to use. See gen_tcp:connect/3,4 for details.

SocketOpts

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

See the options used by gen_tcp(3) and ssl(3)

VerboseMode

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

Profile

When started stand_alone only the pid can be used.

UnixSocket

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

Note:

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

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

The current implementation assumes the requests to the same host, port combination will use the same socket options.

ssl_verify_host_options(WildcardHostName) -> list()

Types:
WildCardHostName = boolean()

Returns ssl options which can be used to verify the host, uses public_key: cacerts_get() to read CA certificates and if WildcardHostName is true adds the hostname check from public_key: public_key: pkix_verify_hostname_match_fun(https) to the options.

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

Types:
  SetCookieHeaders = [HttpHeader]
  Where field = "set-cookie"
  HttpHeaders = {Field :: [byte()], Value :: binary() | iolist()}
  Url = term()
  Profile = atom() | pid()
  When started stand-alone only the pid can be used.
  Reason = term()

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

stream_next(Pid) -> ok

Types:
  Pid = pid()
  As received in the stream_start message

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

which_cookies() -> [CookieStores]
which_cookies(Profile) -> [CookieStores]

Types:
  Profile = atom() | pid()
  When started stand-alone only the pid can be used.
  CookieStores = {cookies, Cookies} | {session_cookies, Cookies}
  Cookies = [term()]

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

which_sessions() -> SessionInfo
which_sessions(Profile) -> SessionInfo

Types:
  Profile = atom() | pid()
  When started stand-alone only the pid can be used.
SessionInfo = {GoodSession, BadSessions, NonSessions}
GoodSession = [Session]
BadSessions = NonSessions = [term()]
Session = term()

Internal representation of a session.

This function is intended for debugging only. It produces a slightly processed dump of the session database. The first list of the session information tuple will contain session information on an internal format. The last two lists of the session information tuple should always be empty if the code is working as intended. If no profile is specified, the default profile is used.

SEE ALSO
RFC 2616, inets(3), gen_tcp(3), ssl(3)
An implementation of an HTTP 1.1 compliant web server, as defined in RFC 2616. Provides web server start options, administrative functions, and an Erlang callback API.

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

- **boolean()** = true | false
- **string()** = list of ASCII characters
- **path()** = string() representing a file or a directory path
- **ip_address()** = {N1,N2,N3,N4} % IPv4 | {K1,K2,K3,K4,K5,K6,K7,K8} % IPv6
- **hostname()** = string() representing a host, for example, "foo.bar.com"
- **property()** = atom()

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

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

- `{proplist_file, path()}

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

**Note:**
Note support for legacy configuration file with Apache syntax is dropped in OTP-23.

**Mandatory Properties**

- `{port, integer()}

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

- `{server_name, string()}

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

- `{server_root, path()}

  Defines the home directory of the server, where log files, and so on, can be stored. Relative paths specified in other properties refer to this directory.
{document_root, path()}
  Defines the top directory for the documents that are available on the HTTP server.

Communication Properties

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

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

{socket_type, ip_comm | {ip_comm, Config::proplist()} | {ssl, Config::proplist()}}
  For ip_comm configuration options, see gen_tcp:listen/2, some options that are used internally by httpd cannot
  be set.
  For SSL configuration options, see ssl:listen/2.
  Default is ip_comm.

[Note:]

OTP-25 deprecates the communication properties {socket_type, ip_comm | (ip_comm,
  Config::proplist()) | {essl, Config::proplist()}) replacing it by {socket_type,
  ip_comm | (ip_comm, Config::proplist()) | {ssl, Config::proplist()}).

{ipfamily, inet | inet6}
  Default is inet, legacy option inet6fb4 no longer makes sense and will be translated to inet.

{minimum_bytes_per_second, integer()}
  If given, sets a minimum of bytes per second value for connections.
  If the value is unreached, the socket closes for that connection.
  The option is good for reducing the risk of "slow DoS" attacks.

Erlang Web Server API Modules

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

Limit properties

{customize, atom()}
  A callback module to customize the inets HTTP servers behaviour see httpd_custom_api

{disable_chunked_transfer_encoding_send, boolean()}
  Allows you to disable chunked transfer-encoding when sending a response to an HTTP/1.1 client. Default is false.
Instructs the server whether to use persistent connections when the client claims to be HTTP/1.1 compliant. Default is true.

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

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

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

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

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

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

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

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

MimeType = string() and Extension = string(). Files delivered to the client are MIME typed according to RFC 1590. File suffixes are mapped to MIME types before file delivery. The mapping between file suffixes and MIME types can be specified in the property list.

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

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

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

Defines the look of the value of the server header.

Example: Assuming the version of Inets is 5.8.1, the server header string can look as follows for the different values of server-tokens:
none

"" % A Server: header will not be generated

prod

"inets"

major

"inets/5"

minor

"inets/5.8"

minimal

"inets/5.8.1"

os

"inets/5.8.1 (unix)"

full

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

{private, "foo/bar"}

"foo/bar"

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

{logger, Options::list()}

Currently only one option is supported:

{error, ServerID::atom()}

Produces logger events on logger level error under the hierarchical logger domain: [otp, inets, httpd, ServerID, error] The built in logger formatting function produces log entries from the error reports:

```erlang
#{server_name => string(),
    protocol => internal | 'TCP' | 'TLS' | 'HTTP',
    transport => "TCP" | "TLS", %% Present when protocol = 'HTTP'
    uri => string(), %% Present when protocol = 'HTTP' and URI is valid
    peer => inet:peername(),
    host => inet:hostname(),
    reason => term()}
```

An example of a log entry with only default settings of logger

```erlang
=ERROR REPORT==== 9-Oct-2019::09:33:27.350235 ===
Server: My Server
Protocol: HTTP
Transport: TLS
   URI: /not there
   Host: 127.0.1.1:80
   Peer: 127.0.0.1:45253
Reason: [{statuscode,404},{description,"Object Not Found"}]
```

Using this option makes mod_log and mod_disk_log error logs redundant.

Add the filter
to appropriate logger handler to handle the events. For example to write the error log from an httpd server with a ServerID of my_server to a file you can use the following sys.config:

```
{kernel,

{logger,

{handler, http_error_test, logger_std_h,

#{config => #{ file => "log/http_error.log" },

  filters => [[inets_httpd, {fun logger_filters:domain/2,

  {log, equal,

  [otp, inets, httpd, my_server, error]

  }]},

  filter_default => stop ]})})].

```
or if you want to add it to the default logger via an API:

```
logger:add_handler_filter(default,

  inets_httpd,

  {fun logger_filters:domain/2,

  {log, equal,

  [otp, inets, httpd, my_server, error]}

}).
```

```
{log_format, common | combined}
```

Defines if access logs are to be written according to the common log format or the extended common log format. The common format is one line looking like this:

```
remotehost rfc931 authuser [date] "request" status bytes.
```

Here:

remotehost
  Remote.
rfc931
  The remote username of the client (RFC 931).
authuser
  The username used for authentication.
[date]
  Date and time of the request (RFC 1123).
"request"
  The request line as it came from the client (RFC 1945).
status
  The HTTP status code returned to the client (RFC 1945).
bytes
  The content-length of the document transferred.

The combined format is one line looking like this:

```
remotehost rfc931 authuser [date] "request" status bytes "referer" "user_agent"
```

In addition to the earlier:

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

This affects the access logs written by mod_log and mod_disk_log.
[error_log_format, pretty | compact]

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

pretty has a format corresponding to:

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

compact has a format corresponding to:

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

This affects the error logs written by mod_log and mod_disk_log.

### URL Aliasing Properties - Requires mod_alias

{alias, {Alias, RealName}}

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

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


{re_write, {Re, Replacement}}

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

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


{directory_index, [string()]

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

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


### CGI Properties - Requires mod_cgi

{script_alias, {Alias, RealName}}

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

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

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

{script_re_write, {Re, Replacement}}

Re = string() and Replacement = string(). Have the same behavior as property re_write, except that they also mark the target directory as containing CGI scripts. URLs with a path beginning with url-path are mapped to scripts beginning with directory-filename, for example:
Access to http://your.server.org/cgi-bin/17/foo would cause the server to run the script /web/17/cgi-bin/foo.

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

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

MimeType = string() and CgiScript = string(). action adds an action activating a CGI script whenever a file of a certain MIME type is requested. It propagates the URL and file path of the requested document using the standard CGI PATH_INFO and PATH_TRANSLATED environment variables.

Example:

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

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

Example:

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

ESI Properties - Requires mod_esi

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

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

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

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

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

Log Properties - Requires mod_log
[error_log, path()]
Defines the filename of the error log file to be used to log server errors. If the filename does not begin with a slash (/), it is assumed to be relative to the server_root.

[security_log, path()]
Defines the filename of the access log file to be used to log security events. If the filename does not begin with a slash (/), it is assumed to be relative to the server_root.

[transfer_log, path()]
Defines the filename of the access log file to be used to log incoming requests. If the filename does not begin with a slash (/), it is assumed to be relative to the server_root.

**Disk Log Properties - Requires mod_disk_log**

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

[error_disk_log, path()]
Defines the filename of the (disk_log(3)) error log file to be used to log server errors. If the filename does not begin with a slash (/), it is assumed to be relative to the server_root.

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

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

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

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

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

**Authentication Properties - Requires mod_auth**

[directory, {path(), [{property(), term()}]}]]
The properties for directories are as follows:

[allow_from, all | [RegExpHostString]]
Defines a set of hosts to be granted access to a given directory, for example:
The host 123.34.56.11 and all machines on the 150.100.23 subnet are allowed access.

{deny_from, ["123.34.56.11", "150.100.23"]}

The host 123.34.56.11 and all machines on the 150.100.23 subnet are not allowed access.

{auth_type, plain | dets | mnesia}

Sets the type of authentication database that is used for the directory. The key difference between the different methods is that dynamic data can be saved when Mnesia and Dets are used.

{auth_user_file, path()}

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

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

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

{auth_group_file, path()}

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

Example:
```
group1: bob joe ante
```

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

{auth_name, string()}

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

{auth_access_password, string()}

If set to other than "NoPassword", the password is required for all API calls. If the password is set to "DummyPassword", the password must be changed before any other API calls. To secure the authenticating data, the password must be changed after the web server is started. Otherwise it is written in clear text in the configuration file.
[require_user, [string()]]
    Defines users to grant access to a given directory using a secret password.

[require_group, [string()]]
    Defines users to grant access to a given directory using a secret password.

**Security Properties - Requires mod_security**

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

The properties for the security directories are as follows:

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

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

{block_time, integer()}
    Specifies the number of minutes a user is blocked. After this time has passed, the user automatically regains access. Default is 60.

{fail_expire_time, integer()}
    Specifies the number of minutes a failed user authentication is remembered. If a user authenticates after this time has passed, the previous failed authentications are forgotten. Default is 30.

{auth_timeout, integer()}
    Specifies the number of seconds a successful user authentication is remembered. After this time has passed, the authentication is no longer reported. Default is 30.

**Exports**

info(Pid) -> HttpInformation
info(Pid, Properties) -> HttpInformation

Types:
Pid = pid()
Properties = [atom()]
HttpInformation = [CommonOption] | [CommunicationOption] | [ModOption] | [LimitOption] | [AdminOption]
CommonOption = {port, integer() >= 0} | {server_name, string()} | {server_root, Path} | {document_root, Path}
CommunicationOption = {bind_address, inet:ip_address() | inet:hostname() | any} | {profile, atom()} | {socket_type, ip_comm | {ip_comm, ssl:tls_option() | gen_tcp:option()} | {ssl, ssl:tls_option() | gen_tcp:option()}} | {ipfamily, inet | inet6} | {minimum_bytes_per_second, integer()}
ModOption = {modules, atom()}
LimitOption = {customize, atom()} | {disable_chunked_transfer_encoding_send, boolean()} | {keep_alive, boolean()} | {keep_alive_timeout, integer()} | {max_body_size, integer()} | {max_clients, integer()} | {max_header_size, integer()} | {max_content_length, integer()} | {max_uri_size, integer()} | {max_keep_alive_request, integer()} | {max_client_body_chunk, integer()}
AdminOption = {mime_types, [{MimeType :: string(), Extension :: string()} | Path} | {mime_type, string()} | {server_admin, string()} | {server_tokens, none | prod | major | minor | minimal | os | full | {private, string()}} | {logger, Options :: list()} | {log_format, common | combined} | {error_log_format, pretty | compact}

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

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

info(Address, Port) -> HttpInformation
info(Address, Port, Profile) -> HttpInformation
info(Address, Port, Properties) -> HttpInformation
info(Address, Port, Profile, Properties) -> HttpInformation

Types:
Address = inet:ip_address() | any
Port = integer()
Profile = atom()
Properties = [atom()]
Path = file:name_all()
HttpInformation = [CommonOption] | [CommunicationOption] | [ModOption] | [LimitOption] | [AdminOption]
CommonOption = {port, integer() >= 0} | {server_name, string()} | {server_root, Path} | {document_root, Path}
CommunicationOption = {bind_address, inet:ip_address() | inet:hostname() | any} | {profile, atom()} | {socket_type, ip_comm} | {ip_comm, ssl:tls_option() | gen_tcp:option()} | {ssl, ssl:tls_option() | gen_tcp:option()} | {ipfamily, inet | inet6} | {minimum_bytes_per_second, integer()}
ModOption = {modules, atom()}
LimitOption = {customize, atom()} | {disable_chunked_transfer_encoding_send, boolean()} | {keep_alive, boolean()} | {keep_alive_timeout, integer()} | {max_body_size, integer()} | {max_clients, integer()} | {max_header_size, integer()} | {max_content_length, integer()} | {max_uri_size, integer()} | {max_keep_alive_request, integer()} | {max_client_body_chunk, integer()} | {mime_types, [{MimeType :: string(), Extension :: string()}] | Path} | {mime_type, string()} | {server_admin, string()} | {server_tokens, none | prod | major | minor | minimal | os | full} | {private, string()} | {logger, Options :: list()} | {log_format, common | combined}
{error_log_format, pretty | compact}
Fetched information about the HTTP server. When called with only Address and Port, all properties are fetched. When called with a list of specific properties, they are fetched. The available properties are the same as the start options of the server.

**Note:**
The Address must be the IP address and cannot be the hostname.

reload_config(Config, Mode) -> ok | {error, Reason} | no_return()
Types:
  - Config = file:name_all() | [{Option, Value}]
  - Mode = non_disturbing | disturbing | blocked
  - Option = atom()
  - Value = Reason = term()
Reloads the HTTP server configuration without restarting the server. Incoming requests are answered with a temporary down message during the reload time.

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

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

**ERLANG WEB SERVER API DATA TYPES**
The Erlang web server API data types are as follows:

```erlang
ModData = #mod{}
  -record(mod, {
    data = [],
    socket_type = ip_comm,
    socket, config_db,
    method,
    absolute_uri,
    request_uri,
    http_version,
    request_line,
    parsed_header = [],
    entity_body,
    connection
  }).
```
To access the record in your callback-module use:

```erlang
  -include_lib("inets/include/httpd.hrl").
```
The fields of record mod have the following meaning:
data
    Type \{(InteractionKey,InteractionValue)\} is used to propagate data between modules. Depicted interaction_data() in function type declarations.

socket_type
    socket_type() indicates whether it is an IP socket or an ssl socket.

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

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

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

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

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

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

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

parsed_header
    Type \{(HeaderKey,HeaderValue)\}. parsed_header contains all HTTP header fields from the HTTP request stored in a list as key-value tuples. See RFC 2616 for a listing of all header fields. For example, the date field is stored as \{"date","Wed, 15 Oct 1997 14:35:17 GMT"\}. RFC 2616 defines that HTTP is a case-insensitive protocol and the header fields can be in lower case or upper case. httpd ensures that all header field names are in lower case.

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

connection
    true | false. If set to true, the connection to the client is a persistent connection and is not closed when the request is served.

Exports

Module:do(ModData) -> {proceed, OldData} | {proceed, NewData} | {break, NewData} | done
Types:
    OldData = list()
    NewData = [(response, {statusCode, Body})]
When a valid request reaches httpd, it calls do/1 in each module, defined by the configuration option of Module. The function can generate data for other modules or a response that can be sent back to the client.

The field data in ModData is a list. This list is the list returned from the last call to do/1. Body is the body of the HTTP response that is sent back to the client. An appropriate header is appended to the message. StatusCode is the status code of the response, see RFC 2616 for the appropriate values.

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

If Body is returned and equal to {Fun, Arg}, the web server tries apply/2 on Fun with Arg as argument. The web server expects that the function either returns a list (Body) that is an HTTP response, or the atom sent if the HTTP response is sent back to the client. If close is returned from the function, something has gone wrong and the server signals this to the client by closing the connection.

httpd

Module:remove(ConfigDB) -> ok | {error, Reason}
Types:
  ConfigDB = ets_table()
  Reason = term()

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

Module:store({Option, Value}, Config) -> {ok, {Option, NewValue}} | {error, Reason}
Types:
  Line = string()
  Option = property()
  Config = [{Option, Value}]
  Value = term()
Reason = term()
Checks the validity of the configuration options before saving them in the internal database. This function can also have a side effect, that is, setup of necessary extra resources implied by the configuration option. It can also resolve possible dependencies among configuration options by changing the value of the option. This function only needs clauses for the options implemented by this particular callback module.

Exports

parse_query(QueryString) -> QueryList | uri_string:error()
Types:
  QueryString = string()
  QueryList = [{unicode:chardata(), unicode:chardata() | true}]
parse_query/1 parses incoming data to erl and eval scripts (see mod_esi(3)) as defined in the standard URL format, that is, ‘+’ becomes ‘space’ and decoding of hexadecimal characters (%xx).

SEE ALSO
RFC 2616, inets(3), ssl(3)
httpd_custom_api

Erlang module

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

Exports

response_default_headers() -> [Header]
Types:
  Header = {HeaderName :: string(), HeaderValue::string()}
  string:to_lower/1 will be performed on the HeaderName

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

response_header({HeaderName, HeaderValue}) -> {true, Header} | false
Types:
  Header = {HeaderName :: string(), HeaderValue::string()}
  The header name will be in lower case and should not be altered.

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

request_header({HeaderName, HeaderValue}) -> {true, Header} | false
Types:
  Header = {HeaderName :: string(), HeaderValue::string()}
  The header name will be in lower case and should not be altered.

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

Erlang module

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

Exports

deliver(SocketType, Socket, Data) -> Result
Types:
  SocketType = socket_type()
  Socket = socket()
  Data = io_list() | binary()
  Result = socket_closed | void()
deliver/3 sends Data over Socket using the specified SocketType. Socket and SocketType is to be the socket and the socket_type form the mod record as defined in httpd.hrl

peername(SocketType,Socket) -> {Port,IPAddress}
Types:
  SocketType = socket_type()
  Socket = socket()
  Port = integer()
  IPAddress = string()
peername/2 returns the Port and IPAddress of the remote Socket.

resolve() -> HostName
Types:
  HostName = string()
resolve/0 returns the official HostName of the current host.

SEE ALSO

httpd(3)
This module provides the Erlang web server API module programmer with miscellaneous utility functions.

**Note:**

Note the module is only recommended for using with httpd - for other cases it should be considered as deprecated.

**Exports**

`convert_request_date(DateString) -> ErlDate|bad_date`

Types:

- `DateString = string()`
- `ErlDate = calendar:datetime()`

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

`create_etag(FileInfo) -> Etag`

Types:

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

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

`day(NthDayOfWeek) -> DayOfWeek`

Types:

- `NthDayOfWeek = 1-7`
- `DayOfWeek = string()`

`day/1` converts the day of the week (`NthDayOfWeek`) from an integer (1-7) to an abbreviated string, that is: 1 = "Mon", 2 = "Tue", ..., 7 = "Sat".

`lookup(ETSTable,Key) -> Result`

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

Types:

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

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

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

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

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

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

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

message(StatusCode, PhraseArgs, ConfigDB) -> Message

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

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

301
  string(): A URL pointing at the new document position.
400 | 401 | 500
  none (no PhraseArgs).
403 | 404
  string(): A Request-URI as described in RFC 2616.
501
  {Method, RequestURI, HTTPVersion}: The HTTP Method, Request-URI, and HTTP-Version as defined in RFC 2616.
504
  string(): A string describing why the service was unavailable.
month(NthMonth) -> Month
Types:
    NthMonth = 1-12
    Month = string()
month/1 converts the month NthMonth as an integer (1-12) to an abbreviated string, that is:
1 = "Jan", 2 = "Feb", ..., 12 = "Dec".

multi_lookup(ETSTable,Key) -> Result
Types:
    ETSTable = ets_table()
    Key = term()
    Result = [term()]
multi_lookup extracts all \{Key,Value\} tuples from an ETSTable and returns all Values associated with Key in a list.

reason_phrase(StatusCode) -> Description
Types:
    StatusCode = 100 | 200 | 201 | 202 | 204 | 205 | 206 | 300 | 301 | 302 | 303 | 304 | 308 | 400 | 401 | 402 | 403 | 404 | 405 | 406 | 410 | 411 | 412 | 413 | 414 | 415 | 416 | 417 | 500 | 501 | 502 | 503 | 504 | 505
    Description = string()
reason_phrase returns Description of an HTTP 1.1 StatusCode, for example, 200 is "OK" and 201 is "Created". For more information, see RFC 2616.

rfc1123_date() -> RFC1123Date
rfc1123_date(Date) -> RFC1123Date
Types:
    Date = calendar:datetime()
    RFC1123Date = string()
rfc1123_date/0 returns the current date in RFC 1123 format. rfc_date/1 converts the date in the Erlang format to the RFC 1123 date format.

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

split_script_path(RequestLine) -> Split
Types:
    RequestLine = string()
**Split** = not_a_script | {Path, PathInfo, QueryString}

Path = QueryString = PathInfo = string()

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

split_path(RequestLine) -> {Path,QueryStringOrPathInfo}

Types:

```
RequestLine = Path = QueryStringOrPathInfo = string()
```

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

**SEE ALSO**

httpd(3)
mod_alias

Erlang module

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

Exports

default_index(ConfigDB, Path) -> NewPath
Types:
   ConfigDB = config_db()
   Path = NewPath = string()
If Path is a directory, default_index/2, it starts searching for resources or files that are specified in the config directive DirectoryIndex. If an appropriate resource or file is found, it is appended to the end of Path and then returned. Path is returned unaltered if no appropriate file is found or if Path is not a directory. config_db() is the server config file in ETS table format as described in Inets User's Guide.

path(PathData, ConfigDB, RequestURI) -> Path
Types:
   PathData = interaction_data()
   ConfigDB = config_db()
   RequestURI = Path = string()
path/3 returns the file Path in the RequestURI (see RFC 1945). If the interaction data \{real_name, \{Path,AfterPath\}\} has been exported by mod_alias, Path is returned. If no interaction data has been exported, ServerRoot is used to generate a file Path. config_db() and interaction_data() are as defined in Inets User's Guide.

real_name(ConfigDB, RequestURI, Aliases) -> Ret
Types:
   ConfigDB = config_db()
   RequestURI = string()
   Aliases = [{FakeName,RealName}]
   Ret = {ShortPath,Path,AfterPath}
   ShortPath = Path = AfterPath = string()
real_name/3 traverses Aliases, typically extracted from ConfigDB, and matches each FakeName with RequestURI. If a match is found, FakeName is replaced with RealName in the match. The resulting path is split into two parts, ShortPath and AfterPath, as defined in httpd_util:split_path/1. Path is generated from ShortPath, that is, the result from default_index/2 with ShortPath as an argument. config_db() is the server config file in ETS table format as described in Inets User's Guide.

real_script_name(ConfigDB, RequestURI, ScriptAliases) -> Ret
Types:
   ConfigDB = config_db()
   RequestURI = string()
   ScriptAliases = [{FakeName,RealName}]

Ericsson AB. All Rights Reserved.: inets | 49
Ret = (ShortPath,AfterPath) | not_a_script
ShortPath = AfterPath = string()

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

Erlang module

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

Exports

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

Types:

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

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

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

Types:

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

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

define_group(GroupName, Options) -> true | {error,Reason}  
<name>define_group(GroupName, Port, Dir) -> true | {error, Reason}  
define_group(GroupName, Address, Port, Dir) -> true | {error, Reason}  
Types:
  Options = [Option]
  Option = {port,Port} | {addr,Address} | {dir,Directory} | {authPassword,AuthPassword}
  Port = integer()
  Address = {A,B,C,D} | string() | undefined
  Dir = string()
  GroupName = string()
  AuthPassword = string()
  Reason = term()

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

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

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

define_user(UserName,Options) -> true | {error, Reason}  
define_user(UserName, Port, Dir) -> true | {error, Reason}  
define_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()

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

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

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

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

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

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

list_groups/1, list_groups/2, and list_groups/3 each lists all the groups available. If there is an
error, {error, Reason} is returned. When list_groups/1 is called, options Port and Dir are mandatory.
list_group_members(GroupName, Options) -> {ok, Users} | {error, Reason}
list_group_members(GroupName, Port, Dir) -> {ok, Users} | {error, Reason}
list_group_members(GroupName, Address, Port, Dir) -> {ok, Users} | {error, Reason}
Types:
  GroupName = string()
  Options = [Option]
  Option = {port,Port} | {addr,Address} | {dir,Directory} | {authPassword,AuthPassword}
  Port = integer()
  Address = {A,B,C,D} | string() | undefined
  Dir = string()
  Users = list()
  AuthPassword = string()
  Reason = term()
list_group_members/2, list_group_members/3, and list_group_members/4 each lists the members of a specified group. If the group does not exist or there is an error, {error, Reason} is returned. When list_group_members/2 is called, options Port and Dir are mandatory.

list_users(Options) -> {ok, Users} | {error, Reason}
list_users(Port, Dir) -> {ok, Users} | {error, Reason}
list_users(Address, Port, Dir) -> {ok, Users} | {error, Reason}
Types:
  Options = [Option]
  Option = {port,Port} | {addr,Address} | {dir,Directory} | {authPassword,AuthPassword}
  Port = integer()
  Address = {A,B,C,D} | string() | undefined
  Dir = string()
  Users = list()
  AuthPassword = string()
  Reason = atom()
list_users/1, list_users/2, and list_users/3 each returns a list of users in the user database for a specific Port/Dir. When list_users/1 is called, options Port and Dir are mandatory.

update_password(Port, Dir, OldPassword, NewPassword, NewPassword) -> ok | {error, Reason}
update_password(Address,Port, Dir, OldPassword, NewPassword, NewPassword) -> ok | {error, Reason}
Types:
  Port = integer()
  Address = {A,B,C,D} | string() | undefined
  Dir = string()
  GroupName = string()
OldPassword = string()
NewPassword = string()
Reason = term()

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

SEE ALSO
httpd(3), mod_alias(3)
This module defines the Erlang Server Interface (ESI) API. It is a more efficient way of writing Erlang scripts for your Inets web server than writing them as common CGI scripts.

DATA TYPES

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

\[
\text{env()} = \\
\{\text{EnvKey()::atom()}, \text{Value::term()}\}
\]

Currently supported key value pairs

\{server_software, string()\}

Indicates the inets version.

\{server_name, string()\}

The local hostname.

\{gateway_interface, string()\}

Legacy string used in CGI, just ignore.

\{server_protocol, string()\}

HTTP version, currently "HTTP/1.1"

\{server_port, integer()\}

Servers port number.

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

HTTP request method.

\{remote_adress, inet:ip_address()\}

The clients ip address.

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

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

\{script_name, string()\}

Request URI

\{http_LowerCaseHTTPHeaderName, string()\}

example: \{http_content_type, "text/html"\}

Exports

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

Types:

\[\text{SessionID} = \text{term()}\]
Data = string() | io_list() | binary()
Reason = term()

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

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

Note:

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

Exports

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

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

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

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

Env environment data of the request see description above.

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

Note:

Note that if the body is small all data may be delivered in only one chunk and then the callback will be called with {last, Data::binary(), undefined} without getting called with {first, Data::binary()}. The input State is the last returned State, in it the callback can include any data that it needs to keep track of when handling the chunks.
mod_security

Erlang module

Security Audit and Trailing Functionality

Exports

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

Types:

User = string()
Port = integer()
Address = {A,B,C,D} | string() | undefined
Dir = string()
Seconds = integer() | infinity
Reason = no_such_directory

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

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

Types:

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

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

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

Types:

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

list_blocked_users/1, list_blocked_users/2, and list_blocked_users/3 each returns a list of users that are currently blocked from access.
unblock_user(User, Port) -> true | {error, Reason}
unblock_user(User, Address, Port) -> true | {error, Reason}
unblock_user(User, Port, Dir) -> true | {error, Reason}
unblock_user(User, Address, Port, Dir) -> true | {error, Reason}

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

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

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

Exports

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

Types:
   What = atom()
   Port = integer()
   Address = \{A,B,C,D\} | string()
   Dir = string()
   Data = [Info]
   Info = \{Name, Value\}

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

auth_fail
   A failed user authentication.
user_block
   A user is being blocked from access.
user_unblock
   A user is being removed from the block list.

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

Erlang module

This module is deprecated since OTP 23. Use the module uri_string to properly handle URIs, this is the recommended module since OTP 21.

DATA TYPES

Type definitions that are related to URI:

\[
\text{uri_part}() = \text{byte()} \mid \text{binary()} \\
\]

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

For more information about URI, see RFC 3986.

Exports

decode(EncodedPart) -> DecodePart

Types:

- \(\text{EncodedPart} = \text{uri_part}()\) - A possibly percent encoded URI part
- \(\text{DecodePart} = \text{uri_part}()\)

**Warning:**

Do not use will be removed use uri_string:unquote/1 instead

Decodes a possibly percent encoded URI part

encode(DecodedPart) -> EncodedPart

Types:

- \(\text{DecodedPart} = \text{uri_part}()\)
- \(\text{EncodedPart} = \text{uri_part}()\) - Percent encoded URI part

**Warning:**

Do not use will be removed use uri_string:quote/1 instead

Performs percent encoding.