SWI-Prolog HTTP support

Jan Wielemaker
VU University Amsterdam
University of Amsterdam
The Netherlands
E-mail: J.Wielemaker@vu.nl

August 25, 2015

Abstract

This article documents the package HTTP, a series of libraries for accessing data on HTTP servers as well as providing HTTP server capabilities from SWI-Prolog. Both server and client are modular libraries. The server can be operated from the Unix `inetd` super-daemon as well as as a stand-alone server that runs on all platforms supported by SWI-Prolog.

Further reading:

- HOWTO collection
- Tutorial by Anne Ogborn

Contents

1 Introduction

The HTTP (HyperText Transfer Protocol) is the W3C standard protocol for transferring information between a web-client (browser) and a web-server. The protocol is a simple *envelope* protocol where standard name/value pairs in the header are used to split the stream into messages and communicate about the connection-status. Many languages have client and or server libraries to deal with the HTTP protocol, making it a suitable candidate for general purpose client-server applications.

In this document we describe a modular infra-structure to access web-servers from SWI-Prolog and turn Prolog into a web-server.

Acknowledgements

This work has been carried out under the following projects: GARP, MIA, IBROW, KITS and Multi-MediaN The following people have pioneered parts of this library and contributed with bug-report and suggestions for improvements: Anjo Anjewierden, Bert Bredeweg, Wouter Jansweijer, Bob Wielinga, Jacco van Ossenbruggen, Michiel Hildebrandt, Matt Lilley and Keri Harris.

2 The HTTP client libraries

This package provides two packages for building HTTP clients. The first, `http/http_open` is a lightweight library for opening a HTTP URL address as a Prolog stream. It can only deal with the HTTP GET protocol. The second, `http/http_client` is a more advanced library dealing with *keep-alive*, *chunked transfer* and a plug-in mechanism providing conversions based on the MIME content-type.

2.1 library(http/http_open): Simple HTTP client

See also

- `xpath/3`
- `http_get/3`
- `http_post/4`

This library provides a light-weight HTTP client library to get the data from a URL. The functionality of the library can be extended by loading two additional modules that acts as plugins:

library(*http/http_chunked*)
: Loading this library causes `http_open/3` to support chunked transfer encoding.

library(*http/http_header*)
: Loading this library causes `http_open/3` to support the `POST` method in addition to `GET`, `HEAD` and `DELETE`.

library(*http/http_ssl_plugin*)
: Loading this library causes `http_open/3` HTTPS connections. Relevant options for SLL certificate handling are handed to `ssl_context/3`. This plugin is loaded automatically if the scheme `https` is requested using a default SSL context. See the plugin for additional information regarding security.

Here is a simple example to fetch a web-page:

```
?- http_open('http://www.google.com/search?q=prolog', In, []),
   copy_stream_data(In, user_output),
   close(In).
<!doctype html><head><title>prolog - Google Search</title><script>
...
```

The example below fetches the modification time of a web-page. Note that Modified is '' (the empty atom) if the web-server does not provide a time-stamp for the resource. See also `parse_time/2`.

```
modified(URL, Stamp) :-
        http_open(URL, In,
                  [ method(head),
                    header(last_modified, Modified)
                  ]),
        close(In),
        Modified \== '',
        parse_time(Modified, Stamp).
```

http_open(+*URL, -Stream, +Options*) *[det]*

Open the data at the HTTP server as a Prolog stream. *URL* is either an atom specifying a *URL* or a list representing a broken-down *URL* as specified below. After this predicate succeeds the data can be read from *Stream*. After completion this stream must be closed using the built-in Prolog predicate `close/1`. *Options* provides additional options:

authorization(+*Term*)

Send authorization. Currently only supports `basic(User,Password)`. See also `http_set_authorization/2`.

final_url(-*FinalURL*)

Unify *FinalURL* with the final destination. This differs from the original *URL* if the returned head of the original indicates an HTTP redirect (codes 301, 302 or 303). Without a redirect, *FinalURL* is the same as *URL* if *URL* is an atom, or a *URL* constructed from the parts.

header(*Name, -AtomValue*)

If provided, *AtomValue* is unified with the value of the indicated field in the reply header. *Name* is matched case-insensitive and the underscore (_) matches the hyphen (-). Multiple of these options may be provided to extract multiple header fields. If the header is not available *AtomValue* is unified to the empty atom ('').

headers(-*List*)

If provided, *List* is unified with a list of Name-Value pairs corresponding to fields in the reply header. Name and Value follow the same conventions used by the `header(Name,Value)` option.

method(+*Method*)
: One of `get` (default), `head` or `delete`. The `head` message can be used in combination with the `header(Name, Value)` option to access information on the resource without actually fetching the resource itself. The returned stream must be closed immediately.

  If `library(http/http_header)` is loaded, `http_open/3` also supports `post` and `put`. See the `post(Data)` option.

size(-*Size*)
: *Size* is unified with the integer value of `Content-Length` in the reply header.

version(-*Version*)
: *Version* is a *pair* `Major-Minor`, where *Major* and *Minor* are integers representing the HTTP version in the reply header.

status_code(-*Code*)
: If this option is present and *Code* unifies with the HTTP status code, do **not** translate errors (4xx, 5xx) into an exception. Instead, `http_open/3` behaves as if 200 (success) is returned, providing the application to read the error document from the returned stream.

output(-*Out*)
: Unify the output stream with *Out* and do not close it. This can be used to upgrade a connection.

timeout(+*Timeout*)
: If provided, set a timeout on the stream using `set_stream/2`. With this option if no new data arrives within *Timeout* seconds the stream raises an exception. Default is to wait forever (`infinite`).

post(+*Data*)
: Provided if `library(http/http_header)` is also loaded. *Data* is handed to `http_post_data/3`.

proxy(+*Host:Port*)
: Use an HTTP proxy to connect to the outside world. See also `socket:proxy_for_url/3`. This option overrules the proxy specification defined by `socket:proxy_for_url/3`.

proxy(+*Host, +Port*)
: Synonym for `proxy(+Host:Port)`. Deprecated.

proxy_authorization(+*Authorization*)
: Send authorization to the proxy. Otherwise the same as the `authorization` option.

bypass_proxy(+*Boolean*)
: If `true`, bypass proxy hooks. Default is `false`.

request_header(*Name=Value*)
: Additional name-value parts are added in the order of appearance to the HTTP request header. No interpretation is done.

user_agent(+*Agent*)
: Defines the value of the `User-Agent` field of the HTTP header. Default is `SWI-Prolog`.

The hook `http:open_options/2` can be used to provide default options based on the broken-down *URL*. The option `status_code(-Code)` is particularly useful to query **REST**

interfaces that commonly return status codes other than `200` that need to be be processed by the client code.

Arguments

URL is either an atom (url) or a list of *parts*. If this list is provided, it may contain the fields `scheme`, `user`, `password`, `host`, `port`, `path` and `search` (where the argument of the latter is a list of Name(Value) or Name=Value). Only `host` is mandatory. The following example below opens the *URL* `http://www.example.com/my/path?q=Hello%20World&lang=en`. Note that values must **not** be quoted because the library inserts the required quotes.

```
http_open([ host('www.example.com'),
            path('/my/path'),
            search([ q='Hello world',
                     lang=en
                   ])
          ])
```

Errors `existence_error(url, Id)`
See also `ssl_context/3` for SSL related options if `library(http/http_ssl_plugin)` is loaded.

http_set_authorization(+*URL, +Authorization*) *[det]*

Set user/password to supply with URLs that have *URL* as prefix. If *Authorization* is the atom `-`, possibly defined authorization is cleared. For example:

```
?- http_set_authorization('http://www.example.com/private/',
                          basic('John', 'Secret'))
```

To be done Move to a separate module, so `http_get/3`, etc. can use this too.

iostream:open_hook(+*Spec, +Mode, -Stream, -Close, +Options0, -Options*) *[semidet,multifile]*

Hook implementation that makes `open_any/5` support `http` and `https` URLs for `Mode == read`.

http:open_options(+*Parts, -Options*) *[nondet,multifile]*

This hook is used by the HTTP client library to define default options based on the the broken-down request-URL. The following example redirects all trafic, except for localhost over a proxy:

```
:- multifile
    http:open_options/2.

http:open_options(Parts, Options) :-
    option(host(Host), Parts),
```

```
Host \== localhost,
Options = [proxy('proxy.local', 3128)].
```

This hook may return multiple solutions. The returned options are combined using `merge_options/3` where earlier solutions overrule later solutions.

http:write_cookies(+*Out, +Parts, +Options*) *[semidet,multifile]*

Emit a `Cookie:` header for the current connection. *Out* is an open stream to the HTTP server, *Parts* is the broken-down request (see `uri_components/2`) and *Options* is the list of options passed to http_open. The predicate is called as if using `ignore/1`.

See also

- complements `http:update_cookies/3`.
- `library(http/http_cookies)` implements cookie handling on top of these hooks.

http:update_cookies(+*CookieData, +Parts, +Options*) *[semidet,multifile]*

Update the cookie database. *CookieData* is the value of the `Set-Cookie` field, *Parts* is the broken-down request (see `uri_components/2`) and *Options* is the list of options passed to http_open.

See also

- complements http:write_cookies
- `library(http/http_cookies)` implements cookie handling on top of these hooks.

2.2 The `http/http_client` library

The `http/http_client` library provides more powerful access to reading HTTP resources, providing *keep-alive* connections, *chunked* transfer and conversion of the content, such as breaking down *multipart* data, parsing HTML, etc. The library announces itself as providing `HTTP/1.1`.

http_get(+*URL, -Reply, +Options*)

Performs a HTTP GET request on the given URL and then reads the reply using `http_read_data/3`. Defined options are:

connection(*ConnectionType*)

If `close` (default) a new connection is created for this request and closed after the request has completed. If `'Keep-Alive'` the library checks for an open connection on the requested host and port and re-uses this connection. The connection is left open if the other party confirms the keep-alive and closed otherwise.

http_version(*Major-Minor*)

Indicate the HTTP protocol version used for the connection. Default is `1.1`.

proxy(+*Host, +Port*)

Use an HTTP proxy to connect to the outside world.

proxy_authorization(+*Authorization*)

Send authorization to the proxy. Otherwise the same as the `authorization` option.

status_code(-*Code*)

If this option is present and *Code* unifies with the HTTP status code, do *not* translate errors (4xx, 5xx) into an exception. Instead, `http_get/3` behaves as if 200 (success) is returned, providing the application to read the error document from the returned stream.

timeout(+*Timeout*)

If provided, set a timeout on the stream using `set_stream/2`. With this option if no new data arrives within *Timeout* seconds the stream raises an exception. This option also affects data being written by the server: if the client does not process the next block of data (4096 bytes using the default setup) within *Timeout*, the connection is terminated. Default is to wait forever (`infinite`).

user_agent(+*Agent*)

Defines the value of the `User-Agent` field of the HTTP header. Default is `SWI-Prolog (http://www.swi-prolog.org)`.

range(+*Range*)

Ask for partial content. *Range* is a term *Unit*(*From, To*), where *From* is an integer and *To* is either an integer or the atom `end`. HTTP 1.1 only supports *Unit* = `bytes`. E.g., to ask for bytes 1000-1999, use the option `range(bytes(1000,1999))`.

request_header(*Name* = *Value*)

Add a line "*Name*: *Value*" to the HTTP request header. Both name and value are added uninspected and literally to the request header. This may be used to specify accept encodings, languages, etc. Please check the RFC2616 (HTTP) document for available fields and their meaning.

reply_header(*Header*)

Unify *Header* with a list of *Name*=*Value* pairs expressing all header fields of the reply. See `http_read_request/2` for the result format.

Remaining options are passed to `http_read_data/3`.

http_post(+*URL, +In, -Reply, +Options*)

Performs a HTTP POST request on the given URL. It is equivalent to `http_get/3`, except for providing an *input document*, which is posted using `http_post_data/3`.

http_read_data(+*Header, -Data, +Options*)

Read data from an HTTP stream. Normally called from `http_get/3` or `http_post/4`. When dealing with HTTP POST in a server this predicate can be used to retrieve the posted data. *Header* is the parsed header. *Options* is a list of *Name*(*Value*) pairs to guide the translation of the data. The following options are supported:

to(*Target*)

Do not try to interpret the data according to the MIME-type, but return it literally according to *Target*, which is one of:

stream(*Output*)

Append the data to the given stream, which must be a Prolog stream open for writing. This can be used to save the data in a (memory-)file, forward it to process using a pipe, etc.

atom

Return the result as an atom. Though SWI-Prolog has no limit on the size of atoms and provides atom-garbage collection, this options should be used with care.[1]

[1] Currently atom-garbage collection is activated after the creation of 10,000 atoms.

codes
Return the page as a list of character-codes. This is especially useful for parsing it using grammar rules.

content_type(*Type*)
Overrule the `Content-Type` as provided by the HTTP reply header. Intended as a work-around for badly configured servers.

If no `to`(*Target*) option is provided the library tries the registered plug-in conversion filters. If none of these succeed it tries the built-in content-type handlers or returns the content as an atom. The builtin content filters are described below. The provided plug-ins are described in the following sections.

application/x-www-form-urlencoded
This is the default encoding mechanism for POST requests issued by a web-browser. It is broken down to a list of *Name* = *Value* terms.

Finally, if all else fails the content is returned as an atom.

http_post_data(+*Data, +Stream, +ExtraHeader*)
Write an HTTP POST request to *Stream* using data from *Data* and passing the additional extra headers from *ExtraHeader*. *Data* is one of:

html(+*HTMLTokens*)
Send an HTML token string as produced by the library `html_write` described in section section 3.18.

xml(+*XMLTerm*)
Send an XML document created by passing XMLTerm to `xml_write/3`. The MIME type is `text/xml`.

xml(+*Type, +XMLTerm*)
As `xml`(*XMLTerm*), using the provided MIME type.

file(+*File*)
Send the contents of *File*. The MIME type is derived from the filename extension using `file_mime_type/2`.

file(+*Type, +File*)
Send the contents of *File* using the provided MIME type, i.e. claiming the `Content-type` equals *Type*.

codes(+*Codes*)
Same as string(text/plain, Codes).

codes(+*Type, +Codes*)
Send string (list of character codes) using the indicated MIME-type.

cgi_stream(+*Stream, +Len*)
Read the input from *Stream* which, like CGI data starts with a partial HTTP header. The fields of this header are merged with the provided *ExtraHeader* fields. The first *Len* characters of *Stream* are used.

form(+*ListOfParameter*)
Send data of the MIME type `application/x-www-form-urlencoded` as produced by browsers issuing a POST request from an HTML form. *ListOfParameter* is a list of *Name*=*Value* or *Name*(*Value*).

form_data(+*ListOfData*)
Send data of the MIME type `multipart/form-data` as produced by browsers issuing a POST request from an HTML form using `enctype multipart/form-data`. This is a somewhat simplified MIME `multipart/mixed` encoding used by browser forms including file input fields. *ListOfData* is the same as for the *List* alternative described below. Below is an example from the SWI-Prolog Sesame interface. *Repository*, etc. are atoms providing the value, while the last argument provides a value from a file.

```
        ...,
        http_post([ protocol(http),
                    host(Host),
                    port(Port),
                    path(ActionPath)
                  ],
                  form_data([ repository = Repository,
                              dataFormat = DataFormat,
                              baseURI    = BaseURI,
                              verifyData = Verify,
                              data       = file(File)
                            ]),
                  _Reply,
                  []),
        ...,
```

List
If the argument is a plain list, it is sent using the MIME type `multipart/mixed` and packed using `mime_pack/3`. See `mime_pack/3` for details on the argument format.

2.2.1 The MIME client plug-in

This plug-in library `http/http_mime_plugin` breaks multipart documents that are recognised by the `Content-Type: multipart/form-data` or `Mime-Version: 1.0` in the header into a list of *Name* = *Value* pairs. This library deals with data from web-forms using the `multipart/form-data` encoding as well as the FIPA agent-protocol messages.

2.2.2 The SGML client plug-in

This plug-in library `http/http_sgml_plugin` provides a bridge between the SGML/XML/HTML parser provided by `sgml` and the http client library. After loading this hook the following mime-types are automatically handled by the SGML parser.

text/html
Handed to `sgml` using W3C HTML 4.0 DTD, suppressing and ignoring all HTML syntax errors. *Options* is passed to `load_structure/3`.

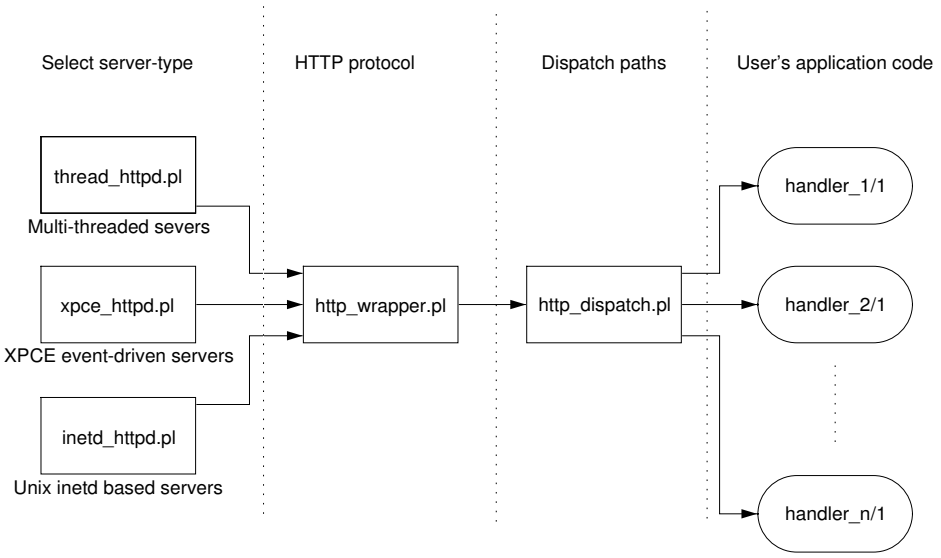

Figure 1: Design of the HTTP server

text/xml
: Handed to `sgml` using dialect `xmlns` (XML + namespaces). *Options* is passed to `load_structure/3`. In particular, `dialect`(*xml*) may be used to suppress namespace handling.

text/x-sgml
: Handled to `sgml` using dialect `sgml`. *Options* is passed to `load_structure/3`.

3 The HTTP server libraries

The HTTP server library consists of two parts obligatory and one optional part. The first deals with connection management and has three different implementation depending on the desired type of server. The second implements a generic wrapper for decoding the HTTP request, calling user code to handle the request and encode the answer. The optional `http_dispatch` module can be used to assign HTTP *locations* (paths) to predicates. This design is summarised in figure 1.

The functional body of the user's code is independent from the selected server-type, making it easy to switch between the supported server types.

3.1 The 'Body'

The server-body is the code that handles the request and formulates a reply. To facilitate all mentioned setups, the body is driven by `http_wrapper/5`. The goal is called with the parsed request (see section 3.11) as argument and `current_output` set to a temporary buffer. Its task is closely related to the task of a CGI script; it must write a header declaring holding at least the `Content-type` field and a body. Here is a simple body writing the request as an HTML table.

```
reply(Request) :-
        format('Content-type: text/html~n~n', []),
```

```
        format('<html>~n', []),
        format('<table border=1>~n'),
        print_request(Request),
        format('~n</table>~n'),
        format('</html>~n', []).

print_request([]).
print_request([H|T]) :-
        H =.. [Name, Value],
        format('<tr><td>~w<td>~w~n', [Name, Value]),
        print_request(T).
```

The infrastructure recognises the header fields described below. Other header lines are passed verbatim to the client. Typical examples are `Set-Cookie` and authentication headers (see section 3.7).

Content-type: *Type* This field is passed to the client and used by the infrastructure to determine the *encoding* to use for the stream. If *type* matches `text/*` or the type matches with `UTF-8` (case insensitive), the server uses UTF-8 encoding. The user may force UTF-8 encoding for arbitrary content types by adding `; charset=UTF-8` to the end of the `Content-type` header.

Transfer-encoding: chunked Causes the server to use *chunked* encoding if the client allows for it. See also section 4 and the `chunked` option in `http_handler/3`.

Connection: close Causes the connection to be closed after the transfer. The default is to keep it open 'Keep-Alive' if possible.

Location: *URL* This header may be combined with the `Status` header to force a *redirect* response to the given *URL*. The message body must be empty. Handling this header is primarily intended for compatibility with the CGI conventions. Prolog code should use `http_redirect/3`.

Status: *Status* This header can be combined with `Location`, where *Status* must be one of 301 (moved), 302 (moved temporary, default) or 303 (see other).

3.1.1 Returning special status codes

Besides returning a page by writing it to the current output stream, the server goal can raise an exception using `throw/1` to generate special pages such as `not_found`, `moved`, etc. The defined exceptions are:

http_reply(+*Reply, +HdrExtra*)
Return a result page using `http_reply/3`. See `http_reply/3` for details.

http_reply(+*Reply*)
Equivalent to `http_reply`(*Reply, []*).

http(*not_modified*)
Equivalent to `http_reply`(*not_modified, []*). This exception is for backward compatibility and can be used by the server to indicate the referenced resource has not been modified since it was requested last time.

In addition, the normal "200 OK" reply status may be overruled by writing a CGI Status header prior to the remainder of the message. This is particularly useful for defining REST APIs. The following handler replies with a "201 Created" header:

```
handle_request(Request) :-
        process_data(Request, Id),      % application predicate
        format('Status: 201~n'),
        format('Content-type: text/plain~n~n'),
        format('Created object as ~q~n', [Id]).
```

3.2 library(http/http_dispatch): Dispatch requests in the HTTP server

This module can be placed between http_wrapper.pl and the application code to associate HTTP *locations* to predicates that serve the pages. In addition, it associates parameters with locations that deal with timeout handling and user authentication. The typical setup is:

```
server(Port, Options) :-
        http_server(http_dispatch,
                    [ port(Port)
                    | Options
                    ]).

:- http_handler('/index.html', write_index, []).

write_index(Request) :-
        ...
```

http_handler(+*Path, :Closure, +Options*) *[det]*

Register *Closure* as a handler for HTTP requests. *Path* is a specification as provided by http_path.pl. If an HTTP request arrives at the server that matches *Path*, *Closure* is called with one extra argument: the parsed HTTP request. *Options* is a list containing the following options:

authentication(+*Type*)

Demand authentication. Authentication methods are pluggable. The library http_authenticate.pl provides a plugin for user/password based Basic HTTP authentication.

chunked

Use Transfer-encoding: chunked if the client allows for it.

content_type(+*Term*)

Specifies the content-type of the reply. This value is currently not used by this library. It enhances the reflexive capabilities of this library through http_current_handler/3.

id(+*Term*)
: Identifier of the handler. The default identifier is the predicate name. Used by `http_location_by_id/2`.

hide_children(+*Bool*)
: If `true` on a prefix-handler (see prefix), possible children are masked. This can be used to (temporary) overrule part of the tree.

prefix
: Call Pred on any location that is a specialisation of *Path*. If multiple handlers match, the one with the longest path is used. *Options* defined with a prefix handler are the default options for paths that start with this prefix. Note that the handler acts as a fallback handler for the tree below it:

```
:- http_handler(/, http_404([index('index.html')]),
                [spawn(my_pool),prefix]).
```

priority(+*Integer*)
: If two handlers handle the same path, the one with the highest priority is used. If equal, the last registered is used. Please be aware that the order of clauses in multifile predicates can change due to reloading files. The default priority is 0 (zero).

spawn(+*SpawnOptions*)
: Run the handler in a seperate thread. If *SpawnOptions* is an atom, it is interpreted as a thread pool name (see `create_thread_pool/3`). Otherwise the options are passed to `http_spawn/2` and from there to `thread_create/3`. These options are typically used to set the stack limits.

time_limit(+*Spec*)
: One of `infinite`, `default` or a positive number (seconds). If `default`, the value from the setting `http:time_limit` is taken. The default of this setting is 300 (5 minutes). See `setting/2`.

Note that `http_handler/3` is normally invoked as a directive and processed using term-expansion. Using term-expansion ensures proper update through `make/0` when the specification is modified. We do not expand when the cross-referencer is running to ensure proper handling of the meta-call.

Errors `existence_error(http_location, Location)`
See also `http_reply_file/3` and `http_redirect/3` are generic handlers to serve files and achieve redirects.

http_delete_handler(+*Spec*) *[det]*
Delete handler for *Spec*. Typically, this should only be used for handlers that are registered dynamically. *Spec* is one of:

id(*Id*)
: Delete a handler with the given id. The default id is the handler-predicate-name.

path(*Path*)
: Delete handler that serves the given path.

http_dispatch(*Request*) *[det]*

Dispatch a *Request* using `http_handler/3` registrations.

http_current_handler(+*Location, :Closure*) *[semidet]*
http_current_handler(-*Location, :Closure*) *[nondet]*

True if *Location* is handled by *Closure*.

http_current_handler(+*Location, :Closure, -Options*) *[semidet]*
http_current_handler(*?Location, :Closure, ?Options*) *[nondet]*

Resolve the current handler and options to execute it.

http_location_by_id(+*ID, -Location*) *[det]*

Find the HTTP *Location* of handler with *ID*. If the setting (see `setting/2`) http:prefix is active, *Location* is the handler location prefixed with the prefix setting. Handler IDs can be specified in two ways:

id(*ID*)

If this appears in the option list of the handler, this it is used and takes preference over using the predicate.

M **:** *PredName*

The module-qualified name of the predicate.

PredName

The unqualified name of the predicate.

Errors `existence_error(http_handler_id, Id)`.

deprecated The predicate `http_link_to_id/3` provides the same functionality with the option to add query parameters or a path parameter.

http_link_to_id(+*HandleID, +Parameters, -HREF*)

HREF is a link on the local server to a handler with given ID, passing the given *Parameters*. This predicate is typically used to formulate a *HREF* that resolves to a handler implementing a particular predicate. The code below provides a typical example. The predicate `user_details/1` returns a page with details about a user from a given id. This predicate is registered as a handler. The DCG `user_link//1` renders a link to a user, displaying the name and calling `user_details/1` when clicked. Note that the location (`root(user_details)`) is irrelevant in this equation and HTTP locations can thus be moved freely without breaking this code fragment.

```
:- http_handler(root(user_details), user_details, []).

user_details(Request) :-
    http_parameters(Request,
                    [ user_id(ID)
                    ]),
    ...

user_link(ID) -->
    { user_name(ID, Name),
```

```
    http_link_to_id(user_details, [id(ID)], HREF)
  },
  html(a([class(user), href(HREF)], Name)).
```

Arguments

Parameters is one of

- `path_postfix(File)` to pass a single value as the last segment of the HTTP location (path). This way of passing a parameter is commonly used in REST APIs.
- A list of search parameters for a `GET` request.

See also `http_location_by_id/2` and `http_handler/3` for defining and specifying handler IDs.

http_reload_with_parameters(+*Request, +Parameters, -HREF*) *[det]*

Create a request on the current handler with replaced search parameters.

http_reply_file(+*FileSpec, +Options, +Request*) *[det]*

Options is a list of

cache(+*Boolean*)
If `true` (default), handle If-modified-since and send modification time.

mime_type(+*Type*)
Overrule mime-type guessing from the filename as provided by `file_mime_type/2`.

static_gzip(+*Boolean*)
If true (default `false`) and, in addition to the plain file, there is a =.gz= file that is not older than the plain file and the client acceps `gzip` encoding, send the compressed file with `Transfer-encoding: gzip`.

unsafe(+*Boolean*)
If `false` (default), validate that *FileSpec* does not contain references to parent directories. E.g., specifications such as `www('../../etc/passwd')` are not allowed.

If caching is not disabled, it processes the request headers `If-modified-since` and `Range`.

throws
- `http_reply(not_modified)`
- `http_reply(file(MimeType, Path))`

http_safe_file(+*FileSpec, +Options*) *[det]*

True if *FileSpec* is considered *safe*. If it is an atom, it cannot be absolute and cannot have references to parent directories. If it is of the form `alias(Sub)`, than Sub cannot have references to parent directories.

Errors
- instantiation_error
- `permission_error(read, file, FileSpec)`

http_redirect(+*How, +To, +Request*) *[det]*

Redirect to a new location. The argument order, using the *Request* as last argument, allows for calling this directly from the handler declaration:

```
:- http_handler(root(.),
                http_redirect(moved, myapp('index.html')),
                []).
```

Arguments

How	is one of `moved`, `moved_temporary` or `see_other`
To	is an atom, a aliased path as defined by `http_absolute_location/3`. or a term `location_by_id(Id)`. If *To* is not absolute, it is resolved relative to the current location.

http_404(+*Options, +Request*) *[det]*

Reply using an "HTTP 404 not found" page. This handler is intended as fallback handler for *prefix* handlers. *Options* processed are:

index(*Location*)

If there is no path-info, redirect the request to *Location* using `http_redirect/3`.

Errors `http_reply(not_found(Path))`

http_switch_protocol(*:Goal, +Options*)

Send an `HTTP 101 Switching Protocols"` reply. After sending the reply, the HTTP library calls `call(Goal, InStream, OutStream)`, where InStream and OutStream are the raw streams to the HTTP client. This allows the communication to continue using an an alternative protocol.

If *Goal* fails or throws an exception, the streams are closed by the server. Otherwise *Goal* is responsible for closing the streams. Note that *Goal* runs in the HTTP handler thread. Typically, the handler should be registered using the `spawn` option if `http_handler/3` or *Goal* must call `thread_create/3` to allow the HTTP worker to return to the worker pool.

The streams use binary (octet) encoding and have their I/O timeout set to the server timeout (default 60 seconds). The predicate `set_stream/2` can be used to change the encoding, change or cancel the timeout.

This predicate interacts with the server library by throwing an exception.

Arguments

Options	is reserved for future extensions. It must be initialised to the empty list ([]).

throws `http_reply(switch_protocol(Goal, Options))`

3.3 library(http/http_dirindex): HTTP directory listings

To be done Provide more options (sorting, selecting columns, hiding files)

This module provides a simple API to generate an index for a physical directory. The index can be customised by overruling the dirindex.css CSS file and by defining additional rules for icons using the hook `http:file_extension_icon/2`.

http_reply_dirindex(+*DirSpec, +Options, +Request*) *[det]*

Provide a directory listing for *Request*, assuming it is an index for the physical directrory Dir. If the request-path does not end with /, first return a moved (301 Moved Permanently) reply.

The calling conventions allows for direct calling from `http_handler/3`.

directory_index(+*Dir, +Options*) **//** *[det]*

Show index for a directory. *Options* processed:

order_by(+*Field*)

Sort the files in the directory listing by *Field*. *Field* is one of `name` (default), `size` or `time`.

order(+*AscentDescent*)

Sorting order. Default is `ascending`. The altenative is `descending`

http:mime_type_icon(+*MimeType, -IconName*) *[nondet,multifile]*

Multi-file hook predicate that can be used to associate icons to files listed by `http_reply_dirindex/3`. The actual icon file is located by `absolute_file_name(icons(IconName), Path, []).`

See also `serve_files_in_directory/2` serves the images.

3.4 library(http/http_files): Serve plain files from a hierarchy

See also `pwp_handler/2` provides similar facilities, where .pwp files can be used to add dynamic behaviour.

Although the SWI-Prolog web-server is intended to serve documents that needed to be computed dynamically, serving plain files is sometimes necessary. This small module combines the functionality of `http_reply_file/3` and `http_reply_dirindex/3` to act as a simple web-server. Such a server can be created using the following code sample, which starts a server at port 8080 that serves files from the current directory ('.'). Note that the handler needs a `prefix` option to specify it must handle all paths that begin with the registed location of the handler.

```
:- use_module(library(http/thread_httpd)).
:- use_module(library(http/http_dispatch)).

:- http_handler(root(.), http_reply_from_files('.', []), [prefix]).

:- initialization
      http_server(http_dispatch, [port(8080)]).
```

http_reply_from_files(+*Dir, +Options, +Request*)

HTTP handler that serves files from the directory *Dir*. This handler uses `http_reply_file/3` to reply plain files. If the request resolves to a directory, it uses the option `indexes` to locate an index file (see below) or uses `http_reply_dirindex/3` to create a listing of the directory.

Options:

indexes(+*List*)

List of files tried to find an index for a directory. The default is ['index.html'].

Note that this handler must be tagged as a `prefix` handler (see `http_handler/3` and module introduction). This also implies that it is possible to override more specific locations in the hierarchy using `http_handler/3` with a longer path-specifier.

Parameters

Dir	is either a directory or an path-specification as used by `absolute_file_name/3`. This option provides great flexibility in (re-)locating the physical files and allows merging the files of multiple physical locations into one web-hierarchy by using multiple `user:file_search_path/2` clauses that define the same alias.

See also The hookable predicate `file_mime_type/2` is used to determine the `Content-type` from the file name.

3.5 library(http/http_session): HTTP Session management

This library defines session management based on HTTP cookies. Session management is enabled simply by loading this module. Details can be modified using `http_set_session_options/1`. By default, this module creates a session whenever a request is processes that is inside the hierarchy defined for session handling (see path option in `http_set_session_options/1`. Automatic creation of a session can be stopped using the option `create(noauto)`. The predicate `http_open_session/2` must be used to create a session if `noauto` is enabled. Sessions can be closed using `http_close_session/1`.

If a session is active, `http_in_session/1` returns the current session and `http_session_assert/1` and friends maintain data about the session. If the session is reclaimed, all associated data is reclaimed too.

Begin and end of sessions can be monitored using `library(broadcast)`. The broadcasted messages are:

http_session(*begin(SessionID,Peer)*)

Broadcasted if a session is started

http_session(*end(SessionId,Peer)*)

Broadcasted if a session is ended. See `http_close_session/1`.

For example, the following calls `end_session(SessionId)` whenever a session terminates. Please note that sessions ends are not scheduled to happen at the actual timeout moment of the session. Instead, creating a new session scans the active list for timed-out sessions. This may change in future versions of this library.

```
:- listen(http_session(end(SessionId, Peer)),
          end_session(SessionId)).
```

http_set_session_options(+*Options*) *[det]*

Set options for the session library. Provided options are:

timeout(+*Seconds*)
Session timeout in seconds. Default is 600 (10 min).

cookie(+*Cookiekname*)
Name to use for the cookie to identify the session. Default `swipl_session`.

path(+*Path*)
Path to which the cookie is associated. Default is `/`. Cookies are only sent if the HTTP request path is a refinement of *Path*.

route(+*Route*)
Set the route name. Default is the unqualified hostname. To cancel adding a route, use the empty atom. See `route/1`.

enabled(+*Boolean*)
Enable/disable session management. Sesion management is enabled by default after loading this file.

create(+*Atom*)
Defines when a session is created. This is one of `auto` (default), which creates a session if there is a request whose path matches the defined session path or `noauto`, in which cases sessions are only created by calling `http_open_session/2` explicitely.

proxy_enabled(+*Boolean*)
Enable/disable proxy session management. Proxy session management associates the *originating* IP address of the client to the session rather than the *proxy* IP address. Default is false.

http_session_option(*?Option*) *[nondet]*

True if *Option* is a current option of the session system.

http_set_session(*Setting*) *[det]*

Overrule a setting for the current session. Currently, the only setting that can be overruled is `timeout`.

Errors `permission_error(set, http_session, Setting)` if setting a setting that is not supported on per-session basis.

http_session_id(-*SessionId*) *[det]*

True if *SessionId* is an identifier for the current session.

Arguments

SessionId	is an atom.

Errors `existence_error(http_session, _)`
See also `http_in_session/1` for a version that fails if there is no session.

http_in_session(-*SessionId*) *[semidet]*
True if *SessionId* is an identifier for the current session. The current session is extracted from `session(ID)` from the current HTTP request (see `http_current_request/1`). The value is cached in a backtrackable global variable `http_session_id`. Using a backtrackable global variable is safe because continuous worker threads use a failure driven loop and spawned threads start without any global variables. This variable can be set from the commandline to fake running a goal from the commandline in the context of a session.

See also `http_session_id/1`

http_open_session(-*SessionID, +Options*) *[det]*
Establish a new session. This is normally used if the create option is set to `noauto`. *Options*:

renew(+*Boolean*)
If `true` (default `false`) and the current request is part of a session, generate a new session-id. By default, this predicate returns the current session as obtained with `http_in_session/1`.

Errors `permission_error(open, http_session, CGI)` if this call is used after closing the CGI header.
See also
- `http_set_session_options/1` to control the `create` option.
- `http_close_session/1` for closing the session.

http_session_asserta(+*Data*) *[det]*
http_session_assert(+*Data*) *[det]*
http_session_retract(*?Data*) *[nondet]*
http_session_retractall(*?Data*) *[det]*
Versions of `assert/1`, `retract/1` and `retractall/1` that associate data with the current HTTP session.

http_session_data(*?Data*) *[nondet]*
True if *Data* is associated using `http_session_assert/1` to the current HTTP session.

Errors `existence_error(http_session,_)`

http_current_session(*?SessionID, ?Data*) *[nondet]*
Enumerate the current sessions and associated data. There are two *Pseudo* data elements:

idle(*Seconds*)
Session has been idle for *Seconds*.

peer(*Peer*)
Peer of the connection.

http_close_session(+*SessionID*) *[det]*
Closes an HTTP session. This predicate can be called from any thread to terminate a session. It uses the `broadcast/1` service with the message below.

```
http_session(end(SessionId, Peer))
```

The broadcast is done **before** the session data is destroyed and the listen-handlers are executed in context of the session that is being closed. Here is an example that destroys a Prolog thread that is associated to a thread:

```
:- listen(http_session(end(SessionId, _Peer)),
          kill_session_thread(SessionID)).

kill_session_thread(SessionID) :-
        http_session_data(thread(ThreadID)),
        thread_signal(ThreadID, throw(session_closed)).
```

Succeed without any effect if *SessionID* does not refer to an active session.

If `http_close_session/1` is called from a handler operating in the current session and the CGI stream is still in state `header`, this predicate emits a `Set-Cookie` to expire the cookie.

Errors `type_error(atom, SessionID)`
See also `listen/2` for acting upon closed sessions

http_session_cookie(-*Cookie*) *[det]*

Generate a random cookie that can be used by a browser to identify the current session. The cookie has the format XXXX-XXXX-XXXX-XXXX[.<route>], where XXXX are random hexadecimal numbers and [.<route>] is the optionally added routing information.

3.6 library(http/http_cors): Enable CORS: Cross-Origin Resource Sharing

See also

- `http://en.wikipedia.org/wiki/Cross-site_scripting` for understanding Cross-site scripting.
- `http://www.w3.org/TR/cors/` for understanding CORS

This small module allows for enabling Cross-Origin Resource Sharing (CORS) for a specific request. Typically, CORS is enabled for API services that you want to have useable from browser client code that is loaded from another domain. An example are the LOD and SPARQL services in ClioPatria.

Because CORS is a security risc (see references), it is disabled by default. It is enabled through the setting http:cors. The value of this setting is a list of domains that are allowed to access the service. Because * is used as a wildcard match, the value [*] allows access from anywhere.

Services for which CORS is relevant must call `cors_enable/0` as part of the HTTP response, as shown below. Note that `cors_enable/0` is a no-op if the setting http:cors is set to the empty list ([]).

```
my_handler(Request) :-
      ....,
      cors_enable,
      reply_json(Response, []).
```

If a site uses a *Preflight* `OPTIONS` request to find the server's capabilities and access politics, `cors_enable/2` can be used to formulate an appropriate reply. For example:

```
my_handler(Request) :-
      option(method(options), Request), !,
      cors_enable(Request,
                  [ methods([get,post,delete])
                  ]),
      format('~n').                                % 200 with empty body
```

cors_enable *[det]*

Emit the HTTP header `Access-Control-Allow-Origin` using domains from the setting http:cors. This this setting is [] (default), nothing is written. This predicate is typically used for replying to API HTTP-request (e.g., replies to an AJAX request that typically serve JSON or XML).

cors_enable(+*Request, +Options*) *[det]*

CORS reply to a *Preflight* `OPTIONS` request. *Request* is the HTTP request. *Options* provides:

methods(+*List*)

List of supported HTTP methods. The default is `GET`, only allowing for read requests.

headers(+*List*)

List of headers the client asks for and we allow. The default is to simply echo what has been requested for.

Both methods and headers may use Prolog friendly syntax, e.g., `get` for a method and `content_type` for a header.

See also `http://www.html5rocks.com/en/tutorials/cors/`

3.7 library(http/http_authenticate): Authenticate HTTP connections using 401 headers

This module provides the basics to validate an HTTP `Authorization` header. User and password information are read from a Unix/Apache compatible password file.

This library provides, in addition to the HTTP authentication, predicates to read and write password files.

http_authenticate(+*Type, +Request, -Fields*)

True if *Request* contains the information to continue according to *Type*. *Type* identifies the required authentication technique:

basic(+*PasswordFile*)

Use HTTP `Basic` authetication and verify the password from *PasswordFile*. *PasswordFile* is a file holding usernames and passwords in a format compatible to Unix and Apache. Each line is record with `:` separated fields. The first field is the username and the second the password *hash*. Password hashes are validated using `crypt/2`.

Successful authorization is cached for 60 seconds to avoid overhead of decoding and lookup of the user and password data.

`http_authenticate/3` just validates the header. If authorization is not provided the browser must be challenged, in response to which it normally opens a user-password dialogue. Example code realising this is below. The exception causes the HTTP wrapper code to generate an HTTP 401 reply.

```
(   http_authenticate(basic(passwd), Request, Fields)
->  true
;   throw(http_reply(authorise(basic, Realm)))
).
```

Arguments

Fields	is a list of fields from the password-file entry. The first element is the user. The hash is skipped.

To be done Should we also cache failures to reduce the risc of DoS attacks?

http_authorization_data(+*AuthorizeText, ?Data*) *[semidet]*

Decode the HTTP `Authorization` header. *Data* is a term

```
Method(User, Password)
```

where Method is the (downcased) authorization method (typically `basic`), User is an atom holding the user name and Password is a list of codes holding the password

http_current_user(+*File, ?User, ?Fields*) *[nondet]*

True when *User* is present in the htpasswd file *File* and *Fields* provides the additional fields.

http_read_passwd_file(+*Path, -Data*) *[det]*

Read a password file. *Data* is a list of terms of the format below, where User is an atom identifying the user, Hash is a string containing the salted password hash and Fields contain additional fields. The string value of each field is converted using `name/2` to either a number or an atom.

```
passwd(User, Hash, Fields)
```

http_write_passwd_file(+*File, +Data:list*) *[det]*

Write password data *Data* to *File*. *Data* is a list of entries as below. See `http_read_passwd_file/2` for details.

```
passwd(User, Hash, Fields)
```

To be done Write to a new file and atomically replace the old one.

3.8 Custom Error Pages

It is possible to create arbitrary error pages for responses generated when a http_reply term is thrown. Currently this is only supported for status 403 (*authentication required*). To do this, instead of throwing `http_reply`(*authorise(Term)*) throw `http_reply`(*authorise(Term), [], Key*), where *Key* is an arbitrary term relating to the page you want to generate. You must then also define a clause of the multifile predicate http:status_page_hook/3:

http:status_page_hook(+*StatusCode, +Key, -CustomHTML*)
StatusCode is the page status code (such as 401), Key is the third argument of the http_reply exception which was thrown, and CustomHTML is a list of HTML tokens. The default page for 401 is generated via this code:

```
        phrase(page([ title('401 Authorization Required')
                    ],
                    [ h1('Authorization Required'),
                      p(['This server could not verify that you ',
                         'are authorized to access the document ',
                         'requested.  Either you supplied the wrong ',
                         'credentials (e.g., bad password), or your ',
                         'browser doesn\'t understand how to supply ',
                         'the credentials required.'
                        ]),
                      \address
                    ]),
               CustomHTML).
```

3.9 library(http/http_openid): OpenID consumer and server library

This library implements the OpenID protocol (`http://openid.net/`). OpenID is a protocol to share identities on the network. The protocol itself uses simple basic HTTP, adding reliability using digitally signed messages.

Steps, as seen from the *consumer* (or *relying partner*).

1. Show login form, asking for `openid_identifier`
2. Get HTML page from `openid_identifier` and lookup `<link rel="openid.server" href="server">`
3. Associate to *server*
4. Redirect browser (302) to server using mode `checkid_setup`, asking to validate the given OpenID.

5. OpenID server redirects back, providing digitally signed conformation of the claimed identity.

6. Validate signature and redirect to the target location.

A **consumer** (an application that allows OpenID login) typically uses this library through `openid_user/3`. In addition, it must implement the hook http_openid:`openid_hook(trusted(OpenId, Server))` to define accepted OpenID servers. Typically, this hook is used to provide a white-list of aceptable servers. Note that accepting any OpenID server is possible, but anyone on the internet can setup a dummy OpenID server that simply grants and signs every request. Here is an example:

```
:- multifile http_openid:openid_hook/1.

http_openid:openid_hook(trusted(_, OpenIdServer)) :-
    (   trusted_server(OpenIdServer)
    ->  true
    ;   throw(http_reply(moved_temporary('/openid/trustedservers')))
    ).

trusted_server('http://www.myopenid.com/server').
```

By default, information who is logged on is maintained with the session using `http_session_assert/1` with the term `openid(Identity)`. The hooks login/logout/logged_in can be used to provide alternative administration of logged-in users (e.g., based on client-IP, using cookies, etc.).

To create a **server**, you must do four things: bind the handlers `openid_server/2` and `openid_grant/1` to HTTP locations, provide a user-page for registered users and define the `grant(Request, Options)` hook to verify your users. An example server is provided in in <plbase>/`doc/packages/examples/demo_openid.pl`

openid_hook(+*Action*) *[multifile]*

Call hook on the OpenID management library. Defined hooks are:

login(+*OpenID*)
: Consider *OpenID* logged in.

logout(+*OpenID*)
: Logout *OpenID*

logged_in(*?OpenID*)
: True if *OpenID* is logged in

grant(+*Request, +Options*)
: Server: Reply positive on OpenID

trusted(+*OpenID, +Server*)
: True if *Server* is a trusted *OpenID* server

ax(*Values*)
: Called if the server provided AX attributes

x_parameter(+*Server, -Name, -Value*)
Called to find additional HTTP parameters to send with the OpenID verify request.

openid_login(+*OpenID*) *[det]*
Associate the current HTTP session with *OpenID*. If another *OpenID* is already associated, this association is first removed.

openid_logout(+*OpenID*) *[det]*
Remove the association of the current session with any *OpenID*

openid_logged_in(-*OpenID*) *[semidet]*
True if session is associated with *OpenID*.

openid_user(+*Request:http_request, -OpenID:url, +Options*) *[det]*
True if *OpenID* is a validated *OpenID* associated with the current session. The scenario for which this predicate is designed is to allow an HTTP handler that requires a valid login to use the transparent code below.

```
handler(Request) :-
      openid_user(Request, OpenID, []),
      ...
```

If the user is not yet logged on a sequence of redirects will follow:

1. Show a page for login (default: page /openid/login), predicate `reply_openid_login/1`)
2. By default, the *OpenID* login page is a form that is submitted to the `verify`, which calls `openid_verify/2`.
3. `openid_verify/2` does the following:
   - Find the *OpenID* claimed identity and server
   - Associate to the *OpenID* server
   - redirects to the *OpenID* server for validation
4. The *OpenID* server will redirect here with the authetication information. This is handled by `openid_authenticate/4`.

Options:

login_url(*Login*)
(Local) URL of page to enter *OpenID* information. Default is the handler for `openid_login_page/1`

See also `openid_authenticate/4` produces errors if login is invalid or cancelled.

openid_login_form(+*ReturnTo, +Options*) **//** *[det]*
Create the OpenID form. This exported as a seperate DCG, allowing applications to redefine /openid/login and reuse this part of the page. *Options* processed:

action(*Action*)
URL of action to call. Default is the handler calling `openid_verify/1`.

buttons(+*Buttons*)
Buttons is a list of `img` structures where the `href` points to an OpenID 2.0 endpoint. These buttons are displayed below the OpenID URL field. Clicking the button sets the URL field and submits the form. Requires Javascript support.

If the `href` is *relative*, clicking it opens the given location after adding 'openid.return_to' and 'stay'.

show_stay(+*Boolean*)
If `true`, show a checkbox that allows the user to stay logged on.

openid_verify(+*Options, +Request*)
Handle the initial login form presented to the user by the relying party (consumer). This predicate discovers the OpenID server, associates itself with this server and redirects the user's browser to the OpenID server, providing the extra openid.X name-value pairs. *Options* is, against the conventions, placed in front of the *Request* to allow for smooth cooperation with `http_dispatch.pl`. *Options* processes:

return_to(+*URL*)
Specifies where the OpenID provider should return to. Normally, that is the current location.

trust_root(+*URL*)
Specifies the `openid.trust_root` attribute. Defaults to the root of the current server (i.e., `http://host[.port]/`).

realm(+*URL*)
Specifies the `openid.realm` attribute. Default is the `trust_root`.

ax(+*Spec*)
Request the exchange of additional attributes from the identity provider. See `http_ax_attributes/2` for details.

The OpenId server will redirect to the `openid.return_to` URL.

throws `http_reply(moved_temporary(Redirect))`

openid_server(*?OpenIDLogin, ?OpenID, ?Server*) *[nondet]*
True if *OpenIDLogin* is the typed id for *OpenID* verified by *Server*.

Arguments

OpenIDLogin	ID as typed by user (canonized)
OpenID	ID as verified by server
Server	URL of the *OpenID* server

openid_current_url(+*Request, -URL*) *[det]* **deprecated**
New code should use `http_public_url/2` with the same semantics.

openid_current_host(*Request, Host, Port*)
Find current location of the server.

deprecated New code should use `http_current_host/4` with the option `global(true)`.

openid_authenticate(+*Request, -Server:url, -OpenID:url, -ReturnTo:url*) *[semidet]*
Succeeds if *Request* comes from the *OpenID* server and confirms that User is a verified *OpenID* user. *ReturnTo* provides the URL to return to.

After `openid_verify/2` has redirected the browser to the *OpenID* server, and the *OpenID* server did its magic, it redirects the browser back to this address. The work is fairly trivial. If `mode` is `cancel`, the OpenId server denied. If `id_res`, the OpenId server replied positive, but we must verify what the server told us by checking the HMAC-SHA signature.

This call fails silently if their is no `openid.mode` field in the request.

throws
- `openid(cancel)` if request was cancelled by the OpenId server
- `openid(signature_mismatch)` if the HMAC signature check failed

openid_server(+*Options, +Request*)
Realise the OpenID server. The protocol demands a POST request here.

openid_grant(+*Request*)
Handle the reply from `checkid_setup_server/3`. If the reply is `yes`, check the authority (typically the password) and if all looks good redirect the browser to ReturnTo, adding the OpenID properties needed by the Relying Party to verify the login.

openid_associate(*?URL, ?Handle, ?Assoc*) *[det]*
Calls `openid_associate/4` as

```
openid_associate(URL, Handle, Assoc, []).
```

openid_associate(+*URL, -Handle, -Assoc, +Options*) *[det]*
openid_associate(*?URL, +Handle, -Assoc, +Options*) *[semidet]*
Associate with an open-id server. We first check for a still valid old association. If there is none or it is expired, we esstablish one and remember it. *Options*:

ns(*URL*)
One of `http://specs.openid.net/auth/2.0` (default) or `http://openid.net/signon/1.1`.

To be done Should we store known associations permanently? Where?

3.10 Get parameters from HTML forms

The library `http/http_parameters` provides two predicates to fetch HTTP request parameters as a type-checked list easily. The library transparently handles both GET and POST requests. It builds on top of the low-level request representation described in section 3.11.

http_parameters(+*Request, ?Parameters*)
The predicate is passes the *Request* as provided to the handler goal by `http_wrapper/5` as well as a partially instantiated lists describing the requested parameters and their types.

Each parameter specification in *Parameters* is a term of the format *Name*(-*Value*, +*Options*). *Options* is a list of option terms describing the type, default, etc. If no options are specified the parameter must be present and its value is returned in *Value* as an atom.

If a parameter is missing the exception `error(existence_error(`*http_parameter, Name*`), _)` is thrown which. If the argument cannot be converted to the requested type, a `error(existence_error(`*Type, Value*`), _)` is raised, where the error context indicates the HTTP parameter. If not caught, the server translates both errors into a `400 Bad request` HTTP message.

Options fall into three categories: those that handle presence of the parameter, those that guide conversion and restrict types and those that support automatic generation of documention. First, the presence-options:

default(*Default***)**
: If the named parameter is missing, *Value* is unified to *Default*.

optional(*true***)**
: If the named parameter is missing, *Value* is left unbound and no error is generated.

list(*Type***)**
: The same parameter may not appear or appear multiple times. If this option is present, `default` and `optional` are ignored and the value is returned as a list. Type checking options are processed on each value.

zero_or_more
: Deprecated. Use `list(`*Type*`)`.

The type and conversion options are given below. The type-language can be extended by providing clauses for the multifile hook http:convert_parameter/3.

;(*Type1, Type2***)**
: Succeed if either *Type1* or *Type2* applies. It allows for checks such as `(nonneg;oneof([infinite]))` to specify an integer or a symbolic value.

oneof(*List***)**
: Succeeds if the value is member of the given list.

length $> N$
: Succeeds if value is an atom of more than N characters.

length $>= N$
: Succeeds if value is an atom of more or than equal to N characters.

length $< N$
: Succeeds if value is an atom of less than N characters.

length $=< N$
: Succeeds if value is an atom of length than or equal to N characters.

atom
: No-op. Allowed for consistency.

string
: Convert value to a string.

between(+*Low, +High*)
Convert value to a number and if either *Low* or *High* is a float, force value to be a float. Then check that the value is in the given range, which includes the boundaries.

boolean
Translate =true=, =yes=, =on= and '1' into =true=; =false=, =no=, =off= and '0' into =false= and raises an error otherwise.

float
Convert value to a float. Integers are transformed into float. Throws a type-error otherwise.

integer
Convert value to an integer. Throws a type-error otherwise.

nonneg
Convert value to a non-negative integer. Throws a type-error of the value cannot be converted to an integer and a domain-error otherwise.

number
Convert value to a number. Throws a type-error otherwise.

The last set of options is to support automatic generation of HTTP API documentation from the sources.[2].

description(+*Atom*)
Description of the parameter in plain text.

group(+*Parameters, +Options*)
Define a logical group of parameters. *Parameters* are processed as normal. *Options* may include a description of the group. Groups can be nested.

Below is an example

```
reply(Request) :-
        http_parameters(Request,
                        [ title(Title, [ optional(true) ]),
                          name(Name,   [ length >= 2 ]),
                          age(Age,     [ between(0, 150) ])
                        ]),
        ...
```

Same as `http_parameters`(*Request, Parameters, []*)

http_parameters(+*Request, ?Parameters, +Options*)
In addition to `http_parameters/2`, the following options are defined.

form_data(-*Data*)
Return the entire set of provided *Name*=*Value* pairs from the GET or POST request. All values are returned as atoms.

[2]This facility is under development in ClioPatria; see `http_help.pl`

attribute_declarations(*:Goal*)

If a parameter specification lacks the parameter options, call `call`(*Goal, +ParamName, -Options*) to find the options. Intended to share declarations over many calls to `http_parameters/3`. Using this construct the above can be written as below.

```
reply(Request) :-
        http_parameters(Request,
                        [ title(Title),
                          name(Name),
                          age(Age)
                        ],
                        [ attribute_declarations(param)
                        ]),
        ...

param(title, [optional(true)]).
param(name,  [length >= 2 ]).
param(age,   [integer]).
```

3.11 Request format

The body-code (see section 3.1) is driven by a *Request*. This request is generated from `http_read_request/2` defined in `http/http_header`.

http_read_request(+*Stream, -Request*)

Reads an HTTP request from *Stream* and unify *Request* with the parsed request. *Request* is a list of *Name*(*Value*) elements. It provides a number of predefined elements for the result of parsing the first line of the request, followed by the additional request parameters. The predefined fields are:

host(*Host*)

If the request contains `Host:` *Host*, Host is unified with the host-name. If *Host* is of the format ⟨*host*⟩:⟨*port*⟩ *Host* only describes ⟨*host*⟩ and a field `port`(*Port*) where *Port* is an integer is added.

input(*Stream*)

The *Stream* is passed along, allowing to read more data or requests from the same stream. This field is always present.

method(*Method*)

Method is one of `get`, `put` or `post`. This field is present if the header has been parsed successfully.

path(*Path*)

Path associated to the request. This field is always present.

peer(*Peer*)

Peer is a term `ip`(*A,B,C,D*) containing the IP address of the contacting host.

port(*Port*)

Port requested. See `host` for details.

request_uri(*RequestURI*)
: This is the untranslated string that follows the method in the request header. It is used to construct the path and search fields of the *Request*. It is provided because reconstructing this string from the path and search fields may yield a different value due to different usage of percent encoding.

search(*ListOfNameValue*)
: Search-specification of URI. This is the part after the `?`, normally used to transfer data from HTML forms that use the '`GET`' protocol. In the URL it consists of a www-form-encoded list of *Name*=*Value* pairs. This is mapped to a list of Prolog *Name*=*Value* terms with decoded names and values. This field is only present if the location contains a search-specification.

http_version(*Major-Minor*)
: If the first line contains the `HTTP/`*Major.Minor* version indicator this element indicate the HTTP version of the peer. Otherwise this field is not present.

cookie(*ListOfNameValue*)
: If the header contains a `Cookie` line, the value of the cookie is broken down in *Name*=*Value* pairs, where the *Name* is the lowercase version of the cookie name as used for the HTTP fields.

set_cookie(*set_cookie(Name, Value, Options)*)
: If the header contains a `SetCookie` line, the cookie field is broken down into the *Name* of the cookie, the *Value* and a list of *Name*=*Value* pairs for additional options such as `expire`, `path`, `domain` or `secure`.

If the first line of the request is tagged with `HTTP/`*Major.Minor*, `http_read_request/2` reads all input upto the first blank line. This header consists of *Name*:*Value* fields. Each such field appears as a term *Name*(*Value*) in the *Request*, where *Name* is canonicalised for use with Prolog. Canonisation implies that the *Name* is converted to lower case and all occurrences of the `-` are replaced by `_`. The value for the `Content-length` fields is translated into an integer.

Here is an example:

```
?- http_read_request(user_input, X).
|: GET /mydb?class=person HTTP/1.0
|: Host: gollem
|:
X = [ input(user),
      method(get),
      search([ class = person
             ]),
      path('/mydb'),
      http_version(1-0),
      host(gollem)
    ].
```

3.11.1 Handling POST requests

Where the HTTP `GET` operation is intended to get a document, using a *path* and possibly some additional search information, the `POST` operation is intended to hand potentially large amounts of data to the server for processing.

The *Request* parameter above contains the term `method`(*post*). The data posted is left on the input stream that is available through the term `input`(*Stream*) from the *Request* header. This data can be read using `http_read_data/3` from the HTTP client library. Here is a demo implementation simply returning the parsed posted data as plain text (assuming `pp/1` pretty-prints the data).

```
reply(Request) :-
        member(method(post), Request), !,
        http_read_data(Request, Data, []),
        format('Content-type: text/plain~n~n', []),
        pp(Data).
```

If the POST is initiated from a browser, content-type is generally either `application/x-www-form-urlencoded` or `multipart/form-data`. The latter is broken down automatically if the plug-in `http/http_mime_plugin` is loaded.

3.12 Running the server

The functionality of the server should be defined in one Prolog file (of course this file is allowed to load other files). Depending on the wanted server setup this 'body' is wrapped into a small Prolog file combining the body with the appropriate server interface. There are three supported server-setups. For most applications we advice the multi-threaded server. Examples of this server architecture are the PlDoc documentation system and the SeRQL Semantic Web server infrastructure.

All the server setups may be wrapped in a *reverse proxy* to make them available from the public web-server as described in section 3.12.7.

- *Using `thread_httpd` for a multi-threaded server*
  This server exploits the multi-threaded version of SWI-Prolog, running the users body code parallel from a pool of worker threads. As it avoids the state engine and copying required in the event-driven server it is generally faster and capable to handle multiple requests concurrently.

  This server is harder to debug due to the involved threading, although the GUI tracer provides reasonable support for multi-threaded applications using the `tspy/1` command. It can provide fast communication to multiple clients and can be used for more demanding servers.

- *Using `inetd_httpd` for server-per-client*
  In this setup the Unix `inetd` user-daemon is used to initialise a server for each connection. This approach is especially suitable for servers that have a limited startup-time. In this setup a crashing client does not influence other requests.

  This server is very hard to debug as the server is not connected to the user environment. It provides a robust implementation for servers that can be started quickly.

3.12.1 Common server interface options

All the server interfaces provide `http_server`(*:Goal, +Options*) to create the server. The list of options differ, but the servers share common options:

port(*?Port*)
: Specify the port to listen to for stand-alone servers. *Port* is either an integer or unbound. If unbound, it is unified to the selected free port.

3.12.2 Multi-threaded Prolog

The `http/thread_httpd.pl` provides the infrastructure to manage multiple clients using a pool of *worker-threads*. This realises a popular server design, also seen in Java Tomcat and Microsoft .NET. As a single persistent server process maintains communication to all clients startup time is not an important issue and the server can easily maintain state-information for all clients.

In addition to the functionality provided by the inetd server, the threaded server can also be used to realise an HTTPS server exploiting the `ssl` library. See option `ssl`(*+SSLOptions*) below.

http_server(*:Goal, +Options*)
: Create the server. *Options* must provide the `port`(*?Port*) option to specify the port the server should listen to. If *Port* is unbound an arbitrary free port is selected and *Port* is unified to this port-number. The server consists of a small Prolog thread accepting new connection on *Port* and dispatching these to a pool of workers. Defined *Options* are:

  port(*?Address*)
  : Address to bind to. *Address* is either a port (integer) or a term *Host*:*Port*. The port may be a variable, causing the system to select a free port and unify the variable with the selected port. See also `tcp_bind/2`.

  workers(*+N*)
  : Defines the number of worker threads in the pool. Default is to use *two* workers. Choosing the optimal value for best performance is a difficult task depending on the number of CPUs in your system and how much resources are required for processing a request. Too high numbers makes your system switch too often between threads or even swap if there is not enough memory to keep all threads in memory, while a too low number causes clients to wait unnecessary for other clients to complete. See also `http_workers/2`.

  timeout(*+SecondsOrInfinite*)
  : Determines the maximum period of inactivity handling a request. If no data arrives within the specified time since the last data arrived the connection raises an exception, the worker discards the client and returns to the pool-queue for a new client. Default is `infinite`, making each worker wait forever for a request to complete. Without a timeout, a worker may wait forever on an a client that doesn't complete its request.

  keep_alive_timeout(*+SecondsOrInfinite*)
  : Maximum time to wait for new activity on *Keep-Alive* connections. Choosing the correct value for this parameter is hard. Disabling Keep-Alive is bad for performance if the clients request multiple documents for a single page. This may —for example– be caused by HTML frames, HTML pages with images, associated CSS files, etc. Keeping a connection open in the threaded model however prevents the thread servicing the client servicing other clients. The default is 5 seconds.

local(+*KBytes*)
Size of the local-stack for the workers. Default is taken from the commandline option.

global(+*KBytes*)
Size of the global-stack for the workers. Default is taken from the commandline option.

trail(+*KBytes*)
Size of the trail-stack for the workers. Default is taken from the commandline option.

ssl(+*SSLOptions*)
Use SSL (Secure Socket Layer) rather than plan TCP/IP. A server created this way is accessed using the `https://` protocol. SSL allows for encrypted communication to avoid others from tapping the wire as well as improved authentication of client and server. The *SSLOptions* option list is passed to `ssl_init/3`. The port option of the main option list is forwarded to the SSL layer. See the `ssl` library for details.

http_server_property(*?Port, ?Property*)
True if *Property* is a property of the HTTP server running at *Port*. Defined properties are:

goal(*:Goal*)
Goal used to start the server. This is often `http_dispatch/1`.

start_time(*?Time*)
Time-stamp when the server was created. See `format_time/3` for creating a human-readable representation.

http_workers(+*Port, ?Workers*)
Query or manipulate the number of workers of the server identified by *Port*. If *Workers* is unbound it is unified with the number of running servers. If it is an integer greater than the current size of the worker pool new workers are created with the same specification as the running workers. If the number is less than the current size of the worker pool, this predicate inserts a number of 'quit' requests in the queue, discarding the excess workers as they finish their jobs (i.e. no worker is abandoned while serving a client).

This can be used to tune the number of workers for performance. Another possible application is to reduce the pool to one worker to facilitate easier debugging.

http_add_worker(+*Port, +Options*)
Add a new worker to the HTTP server for port *Port*. *Options* overrule the default queue options. The following additional options are processed:

max_idle_time(+*Seconds*)
The created worker will automatically terminate if there is no new work within Seconds.

http_stop_server(+*Port, +Options*)
Stop the HTTP server at Port. Halting a server is done *gracefully*, which means that requests being processed are not abandoned. The *Options* list is for future refinements of this predicate such as a forced immediate abort of the server, but is currently ignored.

http_current_worker(*?Port, ?ThreadID*)
True if *ThreadID* is the identifier of a Prolog thread serving *Port*. This predicate is motivated to allow for the use of arbitrary interaction with the worker thread for development and statistics.

http_spawn(*:Goal, +Spec*)

Continue handling this request in a new thread running *Goal*. After `http_spawn/2`, the worker returns to the pool to process new requests. In its simplest form, *Spec* is the name of a thread pool as defined by `thread_pool_create/3`. Alternatively it is an option list, whose options are passed to `thread_create_in_pool/4` if *Spec* contains `pool`(*Pool*) or to `thread_create/3` of the pool option is not present. If the dispatch module is used (see section 3.2), spawning is normally specified as an option to the `http_handler/3` registration.

We recomment the use of thread pools. They allow registration of a set of threads using common characteristics, specify how many can be active and what to do if all threads are active. A typical application may define a small pool of threads with large stacks for computation intensive tasks, and a large pool of threads with small stacks to serve media. The declaration could be the one below, allowing for max 3 concurrent solvers and a maximum backlog of 5 and 30 tasks creating image thumbnails.

```
:- use_module(library(thread_pool)).

:- thread_pool_create(compute, 3,
                      [ local(20000), global(100000), trail(50000),
                        backlog(5)
                      ]).
:- thread_pool_create(media, 30,
                      [ local(100), global(100), trail(100),
                        backlog(100)
                      ]).

:- http_handler('/solve',     solve,     [spawn(compute)]).
:- http_handler('/thumbnail', thumbnail, [spawn(media)]).
```

3.12.3 library(http/http_unix_daemon): Run SWI-Prolog HTTP server as a Unix system daemon

See also The file <swi-home>/doc/packages/examples/http/linux-init-script provides a /etc/init.d script for controlling a server as a normal Unix service.

To be done

- Make it work with SSL
- Cleanup issues wrt. loading and initialization of xpce.

This module provides the logic that is needed to integrate a process into the Unix service (daemon) architecture. It deals with the following aspects, all of which may be used/ignored and configured using commandline options:

- Select the port of the server
- Run the startup of the process as root to perform priviledged tasks and the server itself as unpriviledged user, for example to open ports below 1000.
- Fork and detach from the controlling terminal

- Handle console and debug output using a file and/or the syslog daemon.
- Manage a *pid file*

The typical use scenario is to write a file that loads the following components:

1. The application code, including http handlers (see `http_handler/3`).
2. This library
3. Use an initialization directive to start `http_daemon/0`

In the code below, `load` loads the remainder of the webserver code.

```
:- use_module(library(http/http_unix_daemon)).
:- initialization http_daemon.

:- [load].
```

Now, the server may be started using the command below. See `http_daemon/0` for supported options.

```
% [sudo] swipl -s mainfile.pl -- [option ...]
```

Below are some examples. Our first example is completely silent, running on port 80 as user `www`.

```
% swipl -s mainfile.pl -- --user=www --pidfile=/var/run/http.pid
```

Our second example logs HTTP interaction with the syslog daemon for debugging purposes. Note that the argument to `--debug=` is a Prolog term and must often be escaped to avoid misinterpretation by the Unix shell. The debug option can be repeated to log multiple debug topics.

```
% swipl -s mainfile.pl -- --user=www --pidfile=/var/run/http.pid \
        --debug='http(request)' --syslog=http
```

Broadcasting The library uses `broadcast/1` to allow hooking certain events:

http(*pre_server_start*)
: Run *after fork*, just before starting the HTTP server. Can be used to load additional files or perform additional initialisation, such as starting additional threads. Recall that it is not possible to start threads *before* forking.

http(*post_server_start*)
: Run *after* starting the HTTP server.

http_daemon

Start the HTTP server as a daemon process. This predicate processes the following command-line arguments:

–port=Port Start HTTP server at Port. It requires root permission and the option `--user=User` to open ports below 1000. The default port is 80.

–ip=IP Only listen to the given IP address. Typically used as `--ip=localhost` to restrict access to connections from *localhost* if the server itself is behind an (Apache) proxy server running on the same host.

–debug=Topic Enable debugging Topic. See `debug/3`.

–syslog=Ident Write debug messages to the syslog daemon using Ident

–user=User When started as root to open a port below 1000, this option must be provided to switch to the target user. Three actions are performed as user: open the socket, write the pidfile and setup syslog interaction.

–group=Group May be used in addition to `--user`. If omitted, the login group of the target user is used.

–pidfile=File Write the PID of the daemon process to File.

–output=File Send output of the process to File. By default, all Prolog console output‘ is discarded.

–fork[=Bool] If given as `--no-fork` or `--fork=false`, the process runs in the foreground.

–interactive[=Bool] If `true` (default `false`) implies `--no-fork` and presents the Prolog toplevel after starting the server.

–gtrace=[Bool] Use the debugger to trace `http_daemon/1`.

Other options are converted by `argv_options/3` and passed to `http_server/1`. For example, this allows for:

–workers=Count Set the number of workers for the multi-threaded server.

http_daemon(+*Options*)

Helper that is started from `http_daemon/0`. See `http_daemon/0` for options that are processed.

http_server_hook(+*Options*) *[semidet,multifile]*

Hook that is called to start the HTTP server. This hook must be compatible to `http_server(Handler, Options)`. The default is provided by `start_server/1`.

3.12.4 From (Unix) inetd

All modern Unix systems handle a large number of the services they run through the super-server *inetd*. This program reads `/etc/inetd.conf` and opens server-sockets on all ports defined in this file. As a request comes in it accepts it and starts the associated server such that standard I/O refers to the socket. This approach has several advantages:

- *Simplification of servers*
  Servers don't have to know about sockets and -operations.
- *Centralised authorisation*
  Using *tcpwrappers* simple and effective firewalling of all services is realised.
- *Automatic start and monitor*
  The inetd automatically starts the server 'just-in-time' and starts additional servers or restarts a crashed server according to the specifications.

The very small generic script for handling inetd based connections is in `inetd_httpd`, defining `http_server/1`:

http_server(*:Goal, +Options*)
Initialises and runs `http_wrapper/5` in a loop until failure or end-of-file. This server does not support the *Port* option as the port is specified with the `inetd` configuration. The only supported option is *After*.

Here is the example from `demo_inetd`

```
#!/usr/bin/pl -t main -q -f
:- use_module(demo_body).
:- use_module(inetd_httpd).

main :-
        http_server(reply).
```

With the above file installed in `/home/jan/plhttp/demo_inetd`, the following line in `/etc/inetd` enables the server at port 4001 guarded by *tcpwrappers*. After modifying inetd, send the daemon the `HUP` signal to make it reload its configuration. For more information, please check `inetd.conf(5)`.

```
4001 stream tcp nowait nobody /usr/sbin/tcpd /home/jan/plhttp/demo_inetd
```

3.12.5 MS-Windows

There are rumours that *inetd* has been ported to Windows.

3.12.6 As CGI script

To be done.

3.12.7 Using a reverse proxy

There are three options for public deployment of a service. One is to run it on a dedicated machine on port 80, the standard HTTP port. The machine may be a virtual machine running —for example— under VMWARE or XEN. The (virtual) machine approach isolates security threads and allows for using a standard port. The server can also be hosted on a non-standard port such as 8000, or 8080. Using non-standard ports however may cause problems with intermediate proxy- and/or firewall policies. Isolation can be achieved using a Unix *chroot* environment. Another option, also recommended for *Tomcat* servers, is the use of Apache *reverse proxies*. This causes the main web-server to relay requests below a given URL location to our Prolog based server. This approach has several advantages:

- We can access the server on port 80, just as for a dedicated machine. We do not need a machine though and we only need access to the Apache configuration.
- As Apache is doing the front-line service, the Prolog server is normally protected from malformed HTTP requests that could result in denial of service or otherwise compromise the server. In addition, Apache can provide encodings such as compression to the outside world.

Note that the proxy technology can be combined with isolation methods such as dedicated machines, virtual machines and chroot jails. The proxy can also provide load balancing.

Setting up a reverse proxy The Apache reverse proxy setup is really simple. Ensure the modules `proxy` and `proxy_http` are loaded. Then add two simple rules to the server configuration. Below is an example that makes a PlDoc server on port 4000 available from the main Apache server at port 80.

```
ProxyPass        /pldoc/ http://localhost:4000/pldoc/
ProxyPassReverse /pldoc/ http://localhost:4000/pldoc/
```

Apache rewrites the HTTP headers passing by, but using the above rules it does not examine the content. This implies that URLs embedded in the (HTML) content must use relative addressing. If the locations on the public and Prolog server are the same (as in the example above) it is allowed to use absolute locations. I.e. `/pldoc/search` is ok, but `http://myhost.com:4000/pldoc/search` is *not*. If the locations on the server differ, locations must be relative (i.e. not start with `/`.

This problem can also be solved using the contributed Apache module `proxy_html` that can be instructed to rewrite URLs embedded in HTML documents. In our experience, this is not troublefree as URLs can appear in many places in generated documents. JavaScript can create URLs on the fly, which makes rewriting virtually impossible.

3.13 The wrapper library

The body is called by the module `http/http_wrapper.pl`. This module realises the communication between the I/O streams and the body described in section 3.1. The interface is realised by `http_wrapper/5`:

http_wrapper(*:Goal, +In, +Out, -Connection, +Options*)
Handle an HTTP request where *In* is an input stream from the client, *Out* is an output stream to the client and *Goal* defines the goal realising the body. *Connection* is unified to

`'Keep-alive'` if both ends of the connection want to continue the connection or `close` if either side wishes to close the connection.

This predicate reads an HTTP request-header from *In*, redirects current output to a memory file and then runs `call(Goal, Request)`, watching for exceptions and failure. If *Goal* executes successfully it generates a complete reply from the created output. Otherwise it generates an HTTP server error with additional context information derived from the exception.

`http_wrapper/5` supports the following options:

request(*-Request*)
Return the executed request to the caller.

peer(*+Peer*)
Add peer(Peer) to the request header handed to *Goal*. The format of *Peer* is defined by `tcp_accept/3` from the clib package.

http:request_expansion(*+RequestIn, -RequestOut*)
This *multifile* hook predicate is called just before the goal that produces the body, while the output is already redirected to collect the reply. If it succeeds it must return a valid modified request. It is allowed to throw exceptions as defined in section 3.1.1. It is intended for operations such as mapping paths, deny access for certain requests or manage cookies. If it writes output, these must be HTTP header fields that are added *before* header fields written by the body. The example below is from the session management library (see section 3.5) sets a cookie.

```
        ...,
        format('Set-Cookie: ~w=~w; path=~w~n', [Cookie, SessionID, Path]),
        ...,
```

http_current_request(*-Request*)
Get access to the currently executing request. *Request* is the same as handed to *Goal* of `http_wrapper/5` *after* applying rewrite rules as defined by http:request_expansion/2. Raises an existence error if there is no request in progress.

http_relative_path(*+AbsPath, -RelPath*)
Convert an absolute path (without host, fragment or search) into a path relative to the current page, defined as the path component from the current request (see `http_current_request/1`). This call is intended to create reusable components returning relative paths for easier support of reverse proxies.

If —for whatever reason— the conversion is not possible it simply unifies *RelPath* to *AbsPath*.

3.14 library(http/http_host): Obtain public server location

This library finds the public address of the running server. This can be used to construct URLs that are visible from anywhere on the internet. This module was introduced to deal with OpenID, where a reques is redirected to the OpenID server, which in turn redirects to our server (see `http_openid.pl`).

The address is established from the settings http:public_host and http:public_port if provided. Otherwise it is deduced from the request.

http_public_url(+*Request, -URL*) *[det]*

True when *URL* is an absolute *URL* for the current request. Typically, the login page should redirect to this *URL* to avoid loosing the session.

http_public_host_url(+*Request, -URL*) *[det]*

True when *URL* is the public *URL* at which this server can be contacted. This value is not easy to obtain. See `http_public_host/4` for the hardest part: find the host and port.

http_public_host(*?Request, -Hostname, -Port, +Options*) *[det]*

Current global host and port of the HTTP server. This is the basis to form absolute address, which we need for redirection based interaction such as the OpenID protocol. *Options* are:

global(+*Bool*)

If `true` (default `false`), try to replace a local hostname by a world-wide accessible name.

This predicate performs the following steps to find the host and port:

1. Use the settings `http:public_host` and `http:public_port`
2. Use `X-Forwarded-Host` header, which applies if this server runs behind a proxy.
3. Use the `Host` header, which applies for HTTP 1.1 if we are contacted directly.
4. Use `gethostname/1` to find the host and `http_current_server/2` to find the port.

Arguments

Request	is the current request. If it is left unbound, and the request is needed, it is obtained with `http_current_request/1`.

http_current_host(*?Request, -Hostname, -Port, +Options*) *[det]* **deprecated**

Use `http_public_host/4` (same semantics)

3.15 library(http/http_log): HTTP Logging module

Simple module for logging HTTP requests to a file. Logging is enabled by loading this file and ensure the setting http:logfile is not the empty atom. The default file for writing the log is `httpd.log`. See `library(settings)` for details.

The level of logging can modified using the multifile predicate `http_log:nolog/1` to hide HTTP request fields from the logfile and `http_log:password_field/1` to hide passwords from HTTP search specifications (e.g. `/topsecret?password=secret`).

http_log_stream(-*Stream*) *[semidet]*

True when *Stream* is a stream to the opened HTTP log file. Opens the log file in `append` mode if the file is not yet open. The log file is determined from the setting `http:logfile`. If this setting is set to the empty atom (''), this predicate fails.

If a file error is encountered, this is reported using `print_message/2`, after which this predicate silently fails.

http_log_close(+*Reason*) *[det]*
If there is a currently open HTTP logfile, close it after adding a term `server(Reason, Time).` to the logfile. This call is intended for cooperation with the Unix logrotate facility using the following schema:

- Move logfile (the HTTP server keeps writing to the moved file)
- Inform the server using an HTTP request that calls `http_log_close/1`
- Compress the moved logfile

author Suggested by Jacco van Ossenbruggen

http_log(+*Format, +Args*) *[det]*
Write message from *Format* and *Args* to log-stream. See `format/2` for details. Succeed without side effects if logging is not enabled.

password_field(+*Field*) *[semidet,multifile]*
Multifile predicate that can be defined to hide passwords from the logfile.

nolog(+*HTTPField*) *[multifile]*
Multifile predicate that can be defined to hide request parameters from the request logfile.

3.16 Debugging HTTP servers

The library `http/http_error` defines a hook that decorates uncaught exceptions with a stack-trace. This will generate a *500 internal server error* document with a stack-trace. To enable this feature, simply load this library. Please do note that providing error information to the user simplifies the job of a hacker trying to compromise your server. It is therefore not recommended to load this file by default.

The example program `calc.pl` has the error handler loaded which can be triggered by forcing a divide-by-zero in the calculator.

3.17 Handling HTTP headers

The library `http/http_header` provides primitives for parsing and composing HTTP headers. Its functionality is normally hidden by the other parts of the HTTP server and client libraries. We provide a brief overview of `http_reply/3` which can be accessed from the reply body using an exception as explain in section 3.1.1.

http_reply(+*Type, +Stream, +HdrExtra*)
Compose a complete HTTP reply from the term *Type* using additional headers from *HdrExtra* to the output stream *Stream*. *ExtraHeader* is a list of `Field`(*Value*). *Type* is one of:

html(+*HTML*)
Produce a HTML page using `print_html/1`, normally generated using the `http/html_write` described in section 3.18.

file(+*MimeType, +Path*)
Reply the content of the given file, indicating the given MIME type.

tmp_file(+*MimeType, +Path*)
Similar to `File`(+*MimeType, +Path*), but do not include a modification time header.

stream(+*Stream, +Len*)
Reply using the next *Len* characters from *Stream*. The user must provides the MIME type and other attributes through the *ExtraHeader* argument.

cgi_stream(+*Stream, +Len*)
Similar to `stream`(+*Stream, +Len*), but the data on *Stream* must contain an HTTP header.

moved(+*URL*)
Generate a "301 Moved Permanently" page with the given target *URL*.

moved_temporary(+*URL*)
Generate a "302 Moved Temporary" page with the given target *URL*.

see_other(+*URL*)
Generate a "303 See Other" page with the given target *URL*.

not_found(+*URL*)
Generate a "404 Not Found" page.

forbidden(+*URL*)
Generate a "403 Forbidden" page, denying access without challenging the client.

authorise(+*Method, +Realm*)
Generate a "401 Authorization Required", requesting the client to retry using proper credentials (i.e. user and password).

not_modified
Generate a "304 Not Modified" page, indicating the requested resource has not changed since the indicated time.

server_error(+*Error*)
Generate a "500 Internal server error" page with a message generated from a Prolog exception term (see `print_message/2`).

3.18 The `http/html_write` library

Producing output for the web in the form of an HTML document is a requirement for many Prolog programs. Just using `format/2` is not satisfactory as it leads to poorly readable programs generating poor HTML. This library is based on using DCG rules.

The `http/html_write` structures the generation of HTML from a program. It is an extensible library, providing a *DCG* framework for generating legal HTML under (Prolog) program control. It is especially useful for the generation of structured pages (e.g. tables) from Prolog data structures.

The normal way to use this library is through the DCG `html//1`. This non-terminal provides the central translation from a structured term with embedded calls to additional translation rules to a list of atoms that can then be printed using `print_html/[1,2]`.

html(*:Spec*) **//**
The DCG non-terminal `html//1` is the main predicate of this library. It translates the specification for an HTML page into a list of atoms that can be written to a stream using

`print_html/[1,2]`. The expansion rules of this predicate may be extended by defining the multifile DCG html_write:expand//1. *Spec* is either a single specification or a list of single specifications. Using nested lists is not allowed to avoid ambiguity caused by the atom `[]`

- *Atomic data*
  Atomic data is quoted using `html_quoted//1`.
- *Fmt - Args*
  Fmt and *Args* are used as format-specification and argument list to `format/3`. The result is quoted and added to the output list.
- *\List*
  Escape sequence to add atoms directly to the output list. This can be used to embed external HTML code or emit script output. *List* is a list of the following terms:
  - *Fmt - Args*
    Fmt and *Args* are used as format-specification and argument list to `format/3`. The result is added to the output list.
  - *Atomic*
    Atomic values are added directly to the output list.
- *\Term*
  Invoke the non-terminal *Term* in the calling module. This is the common mechanism to realise abstraction and modularisation in generating HTML.
- *Module:Term*
  Invoke the non-terminal ⟨*Module*⟩:⟨*Term*⟩. This is similar to *\Term* but allows for invoking grammar rules in external packages.
- *&(Entity)*
  Emit `&`⟨*Entity*⟩`;` or `&#`⟨*Entity*⟩`;` if *Entity* is an integer. SWI-Prolog atoms and strings are represented as Unicode. Explicit use of this construct is rarely needed because code-points that are not supported by the output encoding are automatically converted into character-entities.
- `Tag`*(Content)*
  Emit HTML element *Tag* using *Content* and no attributes. *Content* is handed to `html//1`. See section 3.18.4 for details on the automatically generated layout.
- `Tag`*(Attributes, Content)*
  Emit HTML element *Tag* using *Attributes* and *Content*. *Attributes* is either a single attribute of a list of attributes. Each attributes is of the format `Name`(*Value*) or *Name*=*Value*. *Value* is the atomic attribute value but allows for a limited functional notation:
  - $A + B$
    Concatenation of A and B
  - $Format\text{-}Arguments$
    Use `format/3` and emit the result as quoted value.
  - `encode`*(Atom)*
    Use `uri_encoded/3` to create a valid URL query component.
  - `location_by_id`*(ID)*
    HTTP location of the HTTP handler with given ID. See `http_location_by_id/2`.

- *A + List*
  List is handled as a URL 'search' component. The list members are terms of the format *Name* = *Value* or `Name`(*Value*). Values are encoded as in the encode option described above.
- *List*
  Emit SGML multi-valued attributes (e.g., `NAMES`). Each value in list is separated by a space. This is particularly useful for setting multiple `class` attributes on an element. For example:

```
        ...
        span(class([c1,c2]), ...),
```

The example below generates a URL that references the predicate `set_lang/1` in the application with given parameters. The `http_handler/3` declaration binds `/setlang` to the predicate `set_lang/1` for which we provide a very simple implementation. The code between `...` is part of an HTML page showing the english flag which, when pressed, calls `set_lang`(*Request*) where *Request* contains the search parameter `lang` = `en`. Note that the HTTP location (path) `/setlang` can be moved without affecting this code.

```
:- http_handler('/setlang', set_lang, []).

set_lang(Request) :-
        http_parameters(Request,
                        [ lang(Lang, [])
                        ]),
        http_session_retractall(lang(_)),
        http_session_assert(lang(Lang)),
        reply_html_page(title('Switched language'),
                        p(['Switch language to ', Lang])).


        ...
        html(a(href(location_by_id(set_lang) + [lang(en)]),
               img(src('/www/images/flags/en.png')))),
        ...
```

page(*:HeadContent, :BodyContent*) **//**

The DCG non-terminal `page//2` generated a complete page, including the SGML `DOCTYPE` declaration. *HeadContent* are elements to be placed in the `head` element and *BodyContent* are elements to be placed in the `body` element.

To achieve common style (background, page header and footer), it is possible to define DCG non-terminals `head//1` and/or `body//1`. Non-terminal `page//1` checks for the definition of these non-terminals in the module it is called from as well as in the `user` module. If no definition is found, it creates a head with only the *HeadContent* (note that the `title` is obligatory) and a `body` with `bgcolor` set to `white` and the provided *BodyContent*.

Note that further customisation is easily achieved using `html//1` directly as `page//2` is (besides handling the hooks) defined as:

```
page(Head, Body) -->
        html([ \['<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 4.0//EN">\n'],
               html([ head(Head),
                      body(bgcolor(white), Body)
                    ])
             ]).
```

page(*:Contents*) **//**
: This version of the `page/[1,2]` only gives you the SGML `DOCTYPE` and the `HTML` element. *Contents* is used to generate both the head and body of the page.

html_begin(*+Begin*) **//**
: Just open the given element. *Begin* is either an atom or a compound term, In the latter case the arguments are used as arguments to the begin-tag. Some examples:

```
        html_begin(table)
        html_begin(table(border(2), align(center)))
```

This predicate provides an alternative to using the *\Command* syntax in the `html//1` specification. The following two fragments are the same. The preferred solution depends on your preferences as well as whether the specification is generated or entered by the programmer.

```
table(Rows) -->
        html(table([border(1), align(center), width('80%')],
                   [ \table_header,
                     \table_rows(Rows)
                   ])).

% or

table(Rows) -->
        html_begin(table(border(1), align(center), width('80%'))),
        table_header,
        table_rows,
        html_end(table).
```

html_end(*+End*) **//**
: End an element. See `html_begin/1` for details.

3.18.1 Emitting HTML documents

The non-terminal `html//1` translates a specification into a list of atoms and layout instructions. Currently the layout instructions are terms of the format `nl`(*N*), requesting at least *N* newlines. Multiple

consecutive `nl(`*1*`)` terms are combined to an atom containing the maximum of the requested number of newline characters.

To simplify handing the data to a client or storing it into a file, the following predicates are available from this library:

reply_html_page(*:Head, :Body***)**
Same as `reply_html_page`(*default, Head, Body*).

reply_html_page(*+Style, :Head, :Body***)**
Writes an HTML page preceded by an HTTP header as required by `http_wrapper` (CGI-style). Here is a simple typical example:

```
reply(Request) :-
        reply_html_page(title('Welcome'),
                        [ h1('Welcome'),
                          p('Welcome to our ...')
                        ]).
```

The header and footer of the page can be hooked using the grammar-rules user:head//2 and user:body//2. The first argument passed to these hooks is the *Style* argument of `reply_html_page/3` and the second is the 2nd (for `head//2`) or 3rd (for `body//2`) argument of `reply_html_page/3`. These hooks can be used to restyle the page, typically by embedding the real body content in a `div`. E.g., the following code provides a menu on top of each page of that is identified using the style *myapp*.

```
:- multifile
        user:body//2.

user:body(myapp, Body) -->
        html(body([ div(id(top), \application_menu),
                    div(id(content), Body)
                  ])).
```

Redefining the `head` can be used to pull in scripts, but typically `html_requires//1` provides a more modular approach for pulling scripts and CSS-files.

print_html(*+List***)**
Print the token list to the Prolog current output stream.

print_html(*+Stream, +List***)**
Print the token list to the specified output stream

html_print_length(*+List, -Length***)**
When calling `html_print/[1,2]` on *List*, *Length* characters will be produced. Knowing the length is needed to provide the `Content-length` field of an HTTP reply-header.

3.18.2 Repositioning HTML for CSS and javascript links

Modern HTML commonly uses CSS and Javascript. This requires <link> elements in the HTML <head> element or <script> elements in the <body>. Unfortunately this seriously harms re-using HTML DCG rules as components as each of these components may rely on their own style sheets or JavaScript code. We added a 'mailing' system to reposition and collect fragments of HTML. This is implemented by `html_post//2`, `html_receive//1` and `html_receive//2`.

html_post(+*Id, :HTML*) **//** *[det]*

Reposition *HTML* to the receiving *Id*. The `html_post//2` call processes *HTML* using `html//1`. Embedded \-commands are executed by `mailman/1` from `print_html/1` or `html_print_length/2`. These commands are called in the calling context of the `html_post//2` call.

A typical usage scenario is to get required CSS links in the document head in a reusable fashion. First, we define `css//1` as:

```
css(URL) -->
        html_post(css,
                  link([ type('text/css'),
                         rel('stylesheet'),
                         href(URL)
                       ])).
```

Next we insert the *unique* CSS links, in the pagehead using the following call to `reply_html_page/2`:

```
        reply_html_page([ title(...),
                          \html_receive(css)
                        ],
                        ...)
```

html_receive(+*Id*) **//** *[det]*

Receive posted HTML tokens. Unique sequences of tokens posted with `html_post//2` are inserted at the location where `html_receive//1` appears.

See also

- The local predicate `sorted_html//1` handles the output of `html_receive//1`.
- `html_receive//2` allows for post-processing the posted material.

html_receive(+*Id, :Handler*) **//** *[det]*

This extended version of `html_receive//1` causes *Handler* to be called to process all messages posted to the channal at the time output is generated. *Handler* is a grammar rule that is called with three extra arguments.

1. A list of Module:Term, of posted terms. Module is the contest module of html_post and Term is the unmodified term. Members are in the order posted and may contain duplicates.
2. DCG input list. The final output must be produced by a call to `html//1`.

3. DCG output list.

Typically, *Handler* collects the posted terms, creating a term suitable for `html//1` and finally calls `html//1`.

The library predefines the receiver channel `head` at the end of the `head` element for all pages that write the html `head` through this library. The following code can be used anywhere inside an HTML generating rule to demand a javascript in the header:

```
js_script(URL) -->
        html_post(head, script([ src(URL),
                                 type('text/javascript')
                               ], [])).
```

This mechanism is also exploited to add XML namespace (`xmlns`) declarations to the (outer) `html` element using `xhml_ns//2`:

xhtml_ns(*Id, Value*) **//**
Demand an xmlns:id=*Value* in the outer html tag. This uses the `html_post/2` mechanism to post to the `xmlns` channel. Rdfa (`http://www.w3.org/2006/07/SWD/RDFa/syntax/`), embedding RDF in (x)html provides a typical usage scenario where we want to publish the required namespaces in the header. We can define:

```
rdf_ns(Id) -->
        { rdf_global_id(Id:'', Value) },
        xhtml_ns(Id, Value).
```

After which we can use `rdf_ns//1` as a normal rule in `html//1` to publish namespaces from `library(semweb/rdf_db)`. Note that this macro only has effect if the dialect is set to `xhtml`. In `html` mode it is silently ignored.

The required `xmlns` receiver is installed by `html_begin//1` using the `html` tag and thus is present in any document that opens the outer `html` environment through this library.

3.18.3 Adding rules for `html//1`

In some cases it is practical to extend the translations imposed by `html//1`. We used this technique to define translation rules for the output of the SWI-Prolog `sgml` package.

The `html//1` non-terminal first calls the multifile ruleset html_write:expand//1.

html_write:expand(+*Spec*) **//**
Hook to add additional translation rules for `html//1`.

html_quoted(+*Atom*) **//**
Emit the text in *Atom*, inserting entity-references for the SGML special characters `<&>`.

html_quoted_attribute(+*Atom*) **//**
Emit the text in *Atom* suitable for use as an SGML attribute, inserting entity-references for the SGML special characters `<&>"`.

3.18.4 Generating layout

Though not strictly necessary, the library attempts to generate reasonable layout in SGML output. It does this only by inserting newlines before and after tags. It does this on the basis of the multifile predicate html_write:layout/3

html_write:layout(+*Tag, -Open, -Close*)

Specify the layout conventions for the element *Tag*, which is a lowercase atom. *Open* is a term *Pre-Post*. It defines that the element should have at least *Pre* newline characters before and *Post* after the tag. The *Close* specification is similar, but in addition allows for the atom `-`, requesting the output generator to omit the close-tag altogether or `empty`, telling the library that the element has declared empty content. In this case the close-tag is not emitted either, but in addition `html//1` interprets *Arg* in `Tag`(*Arg*) as a list of attributes rather than the content.

A tag that does not appear in this table is emitted without additional layout. See also `print_html/[1,2]`. Please consult the library source for examples.

3.18.5 Examples for using the HTML write library

In the following example we will generate a table of Prolog predicates we find from the SWI-Prolog help system based on a keyword. The primary database is defined by the predicate `predicate/5` We will make hyperlinks for the predicates pointing to their documentation.

```
html_apropos(Kwd) :-
        findall(Pred, apropos_predicate(Kwd, Pred), Matches),
        phrase(apropos_page(Kwd, Matches), Tokens),
        print_html(Tokens).

%       emit page with title, header and table of matches

apropos_page(Kwd, Matches) -->
        page([ title(['Predicates for ', Kwd])
             ],
             [ h2(align(center),
                  ['Predicates for ', Kwd]),
               table([ align(center),
                       border(1),
                       width('80%')
                     ],
                     [ tr([ th('Predicate'),
                            th('Summary')
                          ])
                     | \apropos_rows(Matches)
                     ])
             ]).

%       emit the rows for the body of the table.
```

```
apropos_rows([]) -->
        [].
apropos_rows([pred(Name, Arity, Summary)|T]) -->
        html([ tr([ td(\predref(Name/Arity)),
                    td(em(Summary))
                  ])
             ]),
        apropos_rows(T).

%       predref(Name/Arity)
%
%       Emit Name/Arity as a hyperlink to
%
%               /cgi-bin/plman?name=Name&arity=Arity
%
%       we must do form-encoding for the name as it may contain illegal
%       characters.  www_form_encode/2 is defined in library(url).

predref(Name/Arity) -->
        { www_form_encode(Name, Encoded),
          sformat(Href, '/cgi-bin/plman?name=~w&arity=~w',
                  [Encoded, Arity])
        },
        html(a(href(Href), [Name, /, Arity])).

%       Find predicates from a keyword. '$apropos_match' is an internal
%       undocumented predicate.

apropos_predicate(Pattern, pred(Name, Arity, Summary)) :-
        predicate(Name, Arity, Summary, _, _),
        (   '$apropos_match'(Pattern, Name)
        ->  true
        ;   '$apropos_match'(Pattern, Summary)
        ).
```

3.18.6 Remarks on the `http/html_write` library

This library is the result of various attempts to reach at a more satisfactory and Prolog-minded way to produce HTML text from a program. We have been using Prolog for the generation of web pages in a number of projects. Just using `format/2` never was not a real option, generating error-prone HTML from clumsy syntax. We started with a layer on top of `format/2`, keeping track of the current nesting and thus always capable of properly closing the environment.

DCG based translation however, naturally exploits Prolog's term-rewriting primitives. If generation fails for whatever reason it is easy to produce an alternative document (for example holding an error message).

In a future version we will probably define a `goal_expansion/2` to do compile-time optimisation of the library. Quotation of known text and invocation of sub-rules using the *\RuleSet* and ⟨*Module*⟩:¡RuleSet¿ operators are costly operations in the analysis that can be done at compile-time.

3.19 library(http/js_write): Utilities for including JavaScript

This library is a supplement to `library(http/html_write)` for producing JavaScript fragments. Its main role is to be able to call JavaScript functions with valid arguments constructed from Prolog data. For example, suppose you want to call a JavaScript functions to process a list of names represented as Prolog atoms. This can be done using the call below, while without this library you would have to be careful to properly escape special characters.

```
numbers_script(Names) -->
    html(script(type('text/javascript'),
         [ \js_call('ProcessNumbers'(Names)
         ]),
```

The accepted arguments are described with `js_expression//1`.

js_script(+*Content*) **//** *[det]*

Generate a JavaScript `script` element with the given content.

javascript(+*Content, +Vars, +VarDict, -DOM*) *[det]*

Quasi quotation parser for JavaScript that allows for embedding Prolog variables to substitude *identifiers* in the JavaScript snippet. Parameterizing a JavaScript string is achieved using the JavaScript + operator, which results in concatenation at the client side.

```
    ...,
    js_script({|javascript(Id, Config)||
                $(document).ready(function() {
                   $("#"+Id).tagit(Config);
                 });
               |}),
    ...
```

The current implementation tokenizes the JavaScript input and yields syntax errors on unterminated comments, strings, etc. No further parsing is implemented, which makes it possible to produce syntactically incorrect and partial JavaScript. Future versions are likely to include a full parser, generating syntax errors.

The parser produces a term `\List`, which is suitable for `js_script//1` and `html//1`. Embedded variables are mapped to `\js_expression(Var)`, while the remaining text is mapped to atoms.

To be done Implement a full JavaScript parser. Users should *not* rely on the ability to generate partial JavaScript snippets.

js_call(+*Term*) **//** *[det]*

Emit a call to a Javascript function. The Prolog functor is the name of the function. The arguments are converted from Prolog to JavaScript using `js_arg_list//1`. Please not that Prolog functors can be quoted atom and thus the following is legal:

```
...
html(script(type('text/javascript'),
     [ \js_call('x.y.z'(hello, 42)
     ]),
```

js_new(+*Id, +Term*) **//** *[det]*

Emit a call to a Javascript object declaration. This is the same as:

```
['var ', Id, ' = new ', \js_call(Term)]
```

js_arg_list(+*Expressions:list*) **//** *[det]*

Write javascript (function) arguments. This writes "(", Arg, ..., ")". See `js_expression//1` for valid argument values.

js_expression(+*Expression*) **//** *[det]*

Emit a single JSON argument. *Expression* is one of:

Variable Emitted as Javascript `null`

List Produces a Javascript list, where each element is processed by this library.

`object(Attributes)` Where Attributes is a Key-Value list where each pair can be written as Key-Value, Key=Value or Key(Value), accomodating all common constructs for this used in Prolog. $ { K:V, ... } Same as `object(Attributes)`, providing a more JavaScript-like syntax. This may be useful if the object appears literally in the source-code, but is generally less friendlyto produce as a result from a computation.

Dict Emit a dict as a JSON object using `json_write_dict/3`.

`json(Term)` Emits a term using `json_write/3`.

@(Atom) Emits these constants without quotes. Normally used for the symbols `true`, `false` and `null`, but can also be use for emitting JavaScript symbols (i.e. function- or variable names).

Number Emited literally

`symbol(Atom)` Synonym for @(Atom). Deprecated.

Atom or String Emitted as quoted JavaScript string.

js_arg(+*Expression*) **//** *[semidet]*

Same as `js_expression//1`, but fails if *Expression* is invalid, where `js_expression//1` raises an error.

deprecated New code should use `js_expression//1`.

3.20 library(http/http_path): Abstract specification of HTTP server locations

This module provides an abstract specification of HTTP server locations that is inspired on `absolute_file_name/3`. The specification is done by adding rules to the dynamic multifile predicate `http:location/3`. The speficiation is very similar to `user:file_search_path/2`, but takes an additional argument with options. Currently only one option is defined:

priority(+*Integer*)
: If two rules match, take the one with highest priority. Using priorities is needed because we want to be able to overrule paths, but we do not want to become dependent on clause ordering.

  The default priority is 0. Note however that notably libraries may decide to provide a fall-back using a negative priority. We suggest -100 for such cases.

This library predefines three locations at priority -100: The `icons` and `css` aliases are intended for images and css files and are backed up by file a file-search-path that allows finding the icons and css files that belong to the server infrastructure (e.g., `http_dirindex/2`).

root
: The root of the server. Default is /, but this may be overruled using the setting (see `setting/2`) `http:prefix`

Here is an example that binds `/login` to `login/1`. The user can reuse this application while moving all locations using a new rule for the admin location with the option `[priority(10)]`.

```
:- multifile http:location/3.
:- dynamic   http:location/3.

http:location(admin, /, []).

:- http_handler(admin(login), login, []).

login(Request) :-
        ...
```

http:location(+*Alias, -Expansion, -Options*) *[nondet,multifile]*
: Multifile hook used to specify new HTTP locations. *Alias* is the name of the abstract path. *Expansion* is either a term Alias2(Relative), telling `http_absolute_location/3` to translate *Alias* by first translating Alias2 and then applying the relative path Relative or, *Expansion* is an absolute location, i.e., one that starts with a `/`. *Options* currently only supports the priority of the path. If `http:location/3` returns multiple solutions the one with the highest priority is selected. The default priority is 0.

  This library provides a default for the abstract location `root`. This defaults to the setting http:prefix or, when not available to the path `/`. It is adviced to define all locations (ultimately) relative to `root`. For example, use `root('home.html')` rather than `'/home.html'`.

http_absolute_uri(+*Spec, -URI*) *[det]*

URI is the absolute (i.e., starting with `http://`) *URI* for the abstract specification *Spec*. Use `http_absolute_location/3` to create references to locations on the same server.

To be done Distinguish `http` from `https`

http_absolute_location(+*Spec, -Path, +Options*) *[det]*

Path is the HTTP location for the abstract specification *Spec*. *Options*:

relative_to(*Base*)

Path is made relative to *Base*. Default is to generate absolute URLs.

See also `http_absolute_uri/2` to create a reference that can be used on another server.

http_clean_location_cache

HTTP locations resolved through `http_absolute_location/3` are cached. This predicate wipes the cache. The cache is automatically wiped by `make/0` and if the setting http:prefix is changed.

3.21 library(http/html_head): Automatic inclusion of CSS and scripts links

To be done

- Possibly we should add `img//2` to include images from symbolic path notation.
- It would be nice if the HTTP file server could use our location declarations.

This library allows for abstract declaration of available CSS and Javascript resources and their dependencies using `html_resource/2`. Based on these declarations, html generating code can declare that it depends on specific CSS or Javascript functionality, after which this library ensures that the proper links appear in the HTML head. The implementation is based on mail system implemented by `html_post/2` of library `html_write.pl`.

Declarations come in two forms. First of all http locations are declared using the `http_path.pl` library. Second, `html_resource/2` specifies HTML resources to be used in the `head` and their dependencies. Resources are currently limited to Javascript files (.js) and style sheets (.css). It is trivial to add support for other material in the head. See `html_include//1`.

For usage in HTML generation, there is the DCG rule `html_requires//1` that demands named resources in the HTML head.

3.21.1 About resource ordering

All calls to `html_requires//1` for the page are collected and duplicates are removed. Next, the following steps are taken:

1. Add all dependencies to the set
2. Replace multiple members by 'aggregate' scripts or css files. see `use_agregates/4`.
3. Order all resources by demanding that their dependencies preceede the resource itself. Note that the ordering of resources in the dependency list is **ignored**. This implies that if the order matters the dependency list must be split and only the primary dependency must be added.

3.21.2 Debugging dependencies

Use `?- debug(html(script)).` to see the requested and final set of resources. All declared resources are in `html_resource/3`. The `edit/1` command recognises the names of HTML resources.

3.21.3 Predicates

html_resource(+*About, +Properties*) *[det]*

Register an HTML head resource. *About* is either an atom that specifies an HTTP location or a term Alias(Sub). This works similar to `absolute_file_name/2`. See `http:location_path/2` for details. Recognised properties are:

requires(+*Requirements*)

Other required script and css files. If this is a plain file name, it is interpreted relative to the declared resource. *Requirements* can be a list, which is equivalent to multiple requires properties.

virtual(+*Bool*)

If `true` (default `false`), do not include *About* itself, but only its dependencies. This allows for defining an alias for one or more resources.

ordered(+*Bool*)

Defines that the list of requirements is ordered, which means that each requirement in the list depends on its predecessor.

aggregate(+*List*)

States that *About* is an aggregate of the resources in *List*. This means that if both *About* and one of the elements of *List* appears in the dependencies, *About* is kept and the smaller one is dropped. If there are a number of dependencies on the small members, these are replaced with dependency on the big (aggregate) one, for example, to specify that a big javascript is actually the composition of a number of smaller ones.

mime_type(-*Mime*)

May be specified for non-virtual resources to specify the mime-type of the resource. By default, the mime type is derived from the file name using `file_mime_type/2`.

Registering the same *About* multiple times extends the properties defined for *About*. In particular, this allows for adding additional dependencies to a (virtual) resource.

html_current_resource(*?About*) *[nondet]*

True when *About* is a currently known resource.

html_requires(+*ResourceOrList*) **//** *[det]*

Include *ResourceOrList* and all dependencies derived from it and add them to the HTML `head` using `html_post/2`. The actual dependencies are computed during the HTML output phase by `html_insert_resource//1`.

mime_include(+*Mime, +Path*) **//** *[semidet,multifile]*

Hook called to include a link to an HTML resource of type *Mime* into the HTML head. The

Mime type is computed from *Path* using `file_mime_type/2`. If the hook fails, two built-in rules for `text/css` and `text/javascript` are tried. For example, to include a =.pl= files as a Prolog script, use:

```
:- multifile
    html_head:mime_include//2.

html_head:mime_include(text/'x-prolog', Path) --> !,
    html(script([ type('text/x-prolog'),
                  src(Path)
                ],  [])).
```

3.22 library(http/http_pwp): Serve PWP pages through the HTTP server

To be done

- Support elements in the HTML header that allow controlling the page, such as setting the CGI-header, authorization, etc.
- Allow external styling. Pass through `reply_html_page/2`? Allow filtering the DOM before/after PWP?

This module provides convience predicates to include PWP (Prolog Well-formed Pages) in a Prolog web-server. It provides the following predicates:

`pwp_handler` / *2*

This is a complete web-server aimed at serving static pages, some of which include PWP. This API is intended to allow for programming the web-server from a hierarchy of pwp files, prolog files and static web-pages.

`reply_pwp_page` / *3*

Return a single PWP page that is executed in the context of the calling module. This API is intended for individual pages that include so much text that generating from Prolog is undesirable.

pwp_handler(+*Options, +Request*)

Handle PWP files. This predicate is defined to create a simple HTTP server from a hierarchy of PWP, HTML and other files. The interface is kept compatible with the library(http/http_dispatch). In the typical usage scenario, one needs to define an http location and a file-search path that is used as the root of the server. E.g., the following declarations create a self-contained web-server for files in `/web/pwp/`.

```
user:file_search_path(pwp, '/web/pwp').

:- http_handler(root(.), pwp_handler([path_alias(pwp)]), [prefix]).
```

Options include:

path_alias(+*Alias*)
Search for PWP files as Alias(Path). See `absolute_file_name/3`.

index(+*Index*)
Name of the directory index (pwp) file. This option may appear multiple times. If no such option is provided, `pwp_handler/2` looks for `index.pwp`.

view(+*Boolean*)
If `true` (default is `false`), allow for ?view=source to serve PWP file as source.

index_hook(:*Hook*)
If a directory has no index-file, `pwp_handler/2` calls Hook(PhysicalDir, *Options*, *Request*). If this semidet predicate succeeds, the request is considered handled.

hide_extensions(+*List*)
Hide files of the given extensions. The default is to hide .pl files.

dtd(?*DTD*)
DTD to parse the input file with. If unbound, the generated DTD is returned

Errors permission_error(index, http_location, Location) is raised if the handler resolves to a directory that has no index.
See also `reply_pwp_page/3`

reply_pwp_page(:*File, +Options, +Request*)
Reply a PWP file. This interface is provided to server individual locations from PWP files. Using a PWP file rather than generating the page from Prolog may be desirable because the page contains a lot of text (which is cumbersome to generate from Prolog) or because the maintainer is not familiar with Prolog.

Options supported are:

mime_type(+*Type*)
Serve the file using the given mime-type. Default is text/html.

unsafe(+*Boolean*)
Passed to `http_safe_file/2` to check for unsafe paths.

pwp_module(+*Boolean*)
If `true`, (default `false`), process the PWP file in a module constructed from its canonical absolute path. Otherwise, the PWP file is processed in the calling module.

Initial context:

SCRIPT_NAME
Virtual path of the script.

SCRIPT_DIRECTORY
Physical directory where the script lives

QUERY
Var=Value list representing the query-parameters

REMOTE_USER
If access has been authenticated, this is the authenticated user.

REQUEST_METHOD
: One of `get`, `post`, `put` or `head`

CONTENT_TYPE
: Content-type provided with HTTP POST and PUT requests

CONTENT_LENGTH
: Content-length provided with HTTP POST and PUT requests

While processing the script, the file-search-path pwp includes the current location of the script. I.e., the following will find myprolog in the same directory as where the PWP file resides.

```
pwp:ask="ensure_loaded(pwp(myprolog))"
```

See also `pwp_handler/2`.
To be done complete the initial context, as far as possible from CGI variables. See `http://hoohoo.ncsa.illinois.edu/docs/cgi/env.html`

4 Transfer encodings

The HTTP protocol provides for *transfer encodings*. These define filters applied to the data described by the `Content-type`. The two most popular transfer encodings are `chunked` and `deflate`. The `chunked` encoding avoids the need for a `Content-length` header, sending the data in chunks, each of which is preceded by a length. The `deflate` encoding provides compression.

Transfer-encodings are supported by filters defined as foreign libraries that realise an encoding/decoding stream on top of another stream. Currently there are two such libraries: `http/http_chunked.pl` and `zlib.pl`.

There is an emerging hook interface dealing with transfer encodings. The `http/http_chunked.pl` provides a hook used by `http/http_open.pl` to support chunked encoding in `http_open/3`. Note that both `http_open.pl` *and* `http_chunked.pl` must be loaded for `http_open/3` to support chunked encoding.

4.1 The `http/http_chunked` library

http_chunked_open(*+RawStream, -DataStream, +Options*)
: Create a stream to realise HTTP chunked encoding or decoding. The technique is similar to library(zlib), using a Prolog stream as a filter on another stream. See online documentation at `http://www.swi-prolog.org/` for details.

5 library(http/websocket): WebSocket support

See also RFC 6455, `http://tools.ietf.org/html/rfc6455`
To be done Deal with protocol extensions.

WebSocket is a lightweight message oriented protocol on top of TCP/IP streams. It is typically used as an *upgrade* of an HTTP connection to provide bi-directional communication, but can also be used in isolation over arbitrary (Prolog) streams.

The SWI-Prolog interface is based on *streams* and provides `ws_open/3` to create a *websocket stream* from any Prolog stream. Typically, both an input and output stream are wrapped and then combined into a single object using `stream_pair/3`.

The high-level interface provides `http_upgrade_to_websocket/3` to realise a websocket inside the HTTP server infrastructure and `http_open_websocket/3` as a layer over `http_open/3` to realise a client connection. After establishing a connection, `ws_send/2` and `ws_receive/2` can be used to send and receive messages. The predicate `ws_close/2` is provided to perform the closing handshake and dispose of the stream objects.

http_open_websocket(+*URL*, -*WebSocket*, +*Options*) *[det]*

Establish a client websocket connection. This predicate calls `http_open/3` with additional headers to negotiate a websocket connection. In addition to the options processed by http_open, the following options are recognised:

subprotocols(+*List*)

List of subprotocols that are acceptable. The selected protocol is available as ws_property(*WebSocket*, `subprotocol(Protocol)`.

The following example exchanges a message with the html5rocks.websocket.org echo service:

```
?- URL = 'ws://html5rocks.websocket.org/echo',
   http_open_websocket(URL, WS, []),
   ws_send(WS, text('Hello World!')),
   ws_receive(WS, Reply),
   ws_close(WS, 1000, "Goodbye").
URL = 'ws://html5rocks.websocket.org/echo',
WS = <stream>(0xe4a440,0xe4a610),
Reply = websocket{data:"Hello World!", opcode:text}.
```

Arguments

WebSocket	is a stream pair (see `stream_pair/3`)

http_upgrade_to_websocket(:*Goal*, +*Options*, +*Request*)

Create a websocket connection running `call(Goal, WebSocket)`, where WebSocket is a socket-pair. *Options*:

guarded(+*Boolean*)

If `true` (default), guard the execution of *Goal* and close the websocket on both normal and abnormal termination of *Goal*. If `false`, *Goal* itself is responsible for the created websocket. This can be used to create a single thread that manages multiple websockets using I/O multiplexing.

subprotocols(+*List*)

List of acceptable subprotocols.

timeout(+*TimeOut*)

Timeout to apply to the input stream. Default is `infinite`.

Note that the *Request* argument is the last for cooperation with `http_handler/3`. A simple *echo* server that can be accessed at =/ws/= can be implemented as:

```
:- use_module(library(http/websocket)).
:- use_module(library(http/thread_httpd)).
:- use_module(library(http/http_dispatch)).

:- http_handler(root(ws),
                http_upgrade_to_websocket(echo, []),
                [spawn([])]).

echo(WebSocket) :-
    ws_receive(WebSocket, Message),
    (   Message.opcode == close
    ->  true
    ;   ws_send(WebSocket, Message),
        echo(WebSocket)
    ).
```

throws `switching_protocols(Goal, Options)`. The recovery from this exception causes the HTTP infrastructure to call `call(Goal, WebSocket)`.

See also `http_switch_protocol/2`.

ws_send(+*WebSocket, +Message*) *[det]*

Send a message over a websocket. The following terms are allowed for *Message*:

text(+*Text*)
: Send a text message. *Text* is serialized using `write/1`.

binary(+*Content*)
: As `text(+Text)`, but all character codes produced by *Content* must be in the range [0..255]. Typically, *Content* will be an atom or string holding binary data.

prolog(+*Term*)
: Send a Prolog term as a text message. Text is serialized using `write_canonical/1`.

json(+*JSON*)
: Send the Prolog representation of a *JSON* term using `json_write_dict/2`.

string(+*Text*)
: Same as `text(+Text)`, provided for consistency.

close(+*Code, +Text*)
: Send a close message. *Code* is 1000 for normal close. See websocket documentation for other values.

Dict
: A dict that minimally contains an `opcode` key. Other keys used are:

  `format` : *Format*
  : Serialization format used for *Message*.data. *Format* is one of `string`, `prolog` or `json`. See `ws_receive/3`.

`data` : *Term*
If this key is present, it is serialized according to *Message*.format. Otherwise it is serialized using `write/1`, which implies that string and atoms are just sent verbatim.

Note that `ws_start_message/3` does not unlock the stream. This is done by `ws_send/1`. This implies that multiple threads can use `ws_send/2` and the messages are properly serialized.

To be done Provide serialization details using options.

ws_receive(+*WebSocket, -Message:dict*) *[det]*
ws_receive(+*WebSocket, -Message:dict, +Options*) *[det]*
Receive the next message from *WebSocket*. *Message* is a dict containing the following keys:

`opcode` : *OpCode*
OpCode of the message. This is an atom for known opcodes and an integer for unknown ones. If the peer closed the stream, *OpCode* is bound to `close` and data to the atom `end_of_file`.

`data` : *String*
The data, represented as a string. This field is always present. *String* is the empty string if there is no data in the message.

`rsv` : *RSV*
Present if the *WebSocket RSV* header is not 0. *RSV* is an integer in the range [1..7].

If `ping` message is received and *WebSocket* is a stream pair, `ws_receive/1` replies with a `pong` and waits for the next message.

The predicate `ws_receive/3` processes the following options:

format(+*Format*)
Defines how *text* messages are parsed. *Format* is one of

string
Data is returned as a Prolog string (default)

json
Data is parsed using `json_read_dict/3`, which also receives *Options*.

prolog
Data is parsed using `read_term/3`, which also receives *Options*.

To be done Add a hook to allow for more data formats?

ws_close(+*WebSocket:stream_pair, +Code, +Data*) *[det]*
Close a *WebSocket* connection by sending a `close` message if this was not already sent and wait for the close reply.

Arguments

Code	is the numerical code indicating the close status. This is 16-bit integer. The codes are defined in section *7.4.1. Defined Status Codes* of RFC6455. Notably, 1000 indicates a normal closure.
Data	is currently interpreted as text.

Errors `websocket_error(unexpected_message, Reply)` if the other side did not send a close message in reply.

ws_open(+*Stream, -WSStream, +Options*) *[det]*

Turn a raw TCP/IP (or any other binary stream) into a websocket stream. *Stream* can be an input stream, output stream or a stream pair. *Options* includes

mode(+*Mode*)
One of `server` or `client`. If `client`, messages are sent as *masked*.

buffer_size(+*Count*)
Send partial messages for each *Count* bytes or when flushing the output. The default is to buffer the entire message before it is sent.

close_parent(+*Boolean*)
If `true` (default), closing *WSStream* also closes *Stream*.

subprotocol(+*Protocol*)
Set the subprotocol property of WsStream. This value can be retrieved using `ws_property/2`. *Protocol* is an atom. See also the `subprotocols` option of `http_open_websocket/3` and `http_upgrade_to_websocket/3`.

A typical sequence to turn a pair of streams into a WebSocket is here:

```
...,
Options = [mode(server), subprotocol(chat)],
ws_open(Input, WsInput, Options),
ws_open(Output, WsOutput, Options),
stream_pair(WebSocket, WsInput, WsOutput).
```

ws_property(+*WebSocket, ?Property*) *[nondet]*

True if *Property* is a property *WebSocket*. Defined properties are:

subprotocol(*Protocol*)
Protocol is the negotiated subprotocol. This is typically set as a property of the websocket by `ws_open/3`.

6 library(http/hub): Manage a hub for websockets

To be done The current design does not use threads to perform tasks for multiple hubs. This implies that the design scales rather poorly for hosting many hubs with few users.

This library manages a hub that consists of clients that are connected using a websocket. Messages arriving at any of the websockets are sent to the *event* queue of the hub. In addition, the hub provides a *broadcast* interface. A typical usage scenario for a hub is a *chat server* A scenario for realizing an chat server is:

1. Create a new hub using `hub_create/3`.

2. Create one or more threads that listen to Hub.queues.event from the created hub. These threads can update the shared view of the world. A message is a dict as returned by `ws_receive/2` or a hub control message. Currently, the following control messages are defined:

   hub{error*:Error,* left*:ClientId,* reason*:Reason*}
   A client left us because of an I/O error. *Reason* is `read` or `write` and *Error* is the Prolog I/O exception.

   hub{joined*:ClientId*}
   A new client has joined the chatroom.

   The `thread(s)` can talk to clients using two predicates:

   - `hub_send/2` sends a message to a specific client
   - `hub_broadcast/2` sends a message to all clients of the hub.

A hub consists of (currenty) four message queues and a simple dynamic fact. Threads that are needed for the communication tasks are created on demand and die if no more work needs to be done.

hub_create(+*Name, -Hub, +Options*) *[det]*
Create a new hub. *Hub* is a dict containing the following public information:

Hub `.` `name`
The name of the hub (the *Name* argument)

`queues` `.` `event`
Message queue to which the hub `thread(s)` can listen.

After creating a hub, the application normally creates a thread that listens to *Hub*.queues.event and exposes some mechanisms to establish websockets and add them to the hub using `hub_add/3`.

See also `http_upgrade_to_websocket/3` establishes a websocket from the SWI-Prolog webserver.

current_hub(*?Name, ?Hub*) *[nondet]*
True when there exists a hub *Hub* with *Name*.

hub_add(+*Hub, +WebSocket, ?Id*) *[det]*
Add a *WebSocket* to the hub. *Id* is used to identify this user. It may be provided (as a ground term) or is generated as a UUID.

hub_send(+*ClientId, +Message*) *[det]*
Send message to the indicated *ClientId*.

Arguments

Message	is either a single message (as accepted by `ws_send/2`) or a list of such messages.

hub_broadcast(*+Hub, +Message*) *[det]*

Send *Message* to all websockets associated with *Hub*. Note that this process is *asynchronous*: this predicate returns immediately after putting all requests in a broadcast queue. If a message cannot be delivered due to a network error, the hub is informed through `io_error/3`.

7 Supporting JSON

From `http://json.org`, " JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language."

JSON is interesting to Prolog because using AJAX web technology we can easily created web-enabled user interfaces where we implement the server side using the SWI-Prolog HTTP services provided by this package. The interface consists of three libraries:

- `library(http/json)` provides support for the core JSON object serialization.
- `library(http/json_convert)` converts between the primary representation of JSON terms in Prolog and more application oriented Prolog terms. E.g. `point(X,Y)` vs. `object([x=X,y=Y])`.
- `library(http/http_json)` hooks the conversion libraries into the HTTP client and server libraries.

7.1 json.pl: Reading and writing JSON serialization

author Jan Wielemaker
See also
- `http_json.pl` links JSON to the HTTP client and server modules.
- `json_convert.pl` converts JSON Prolog terms to more comfortable terms.

This module supports reading and writing JSON objects. This library supports two Prolog representations (the *new* representation is only supported in SWI-Prolog version 7 and later):

- The **classical** representation is provided by `json_read/3` and `json_write/3`. This represents a JSON object as `json(NameValueList)`, a JSON string as an atom and the JSON constants `null`, `true` and `false` as @(null), @(true) and @false.
- The **new** representation is provided by `json_read_dict/3` and `json_write_dict/3`. This represents a JSON object as a dict, a JSON string as a Prolog string and the JSON constants using the Prolog atoms `null`, `true` and `false`.

atom_json_term(*+Atom, -JSONTerm, +Options*) *[det]*
atom_json_term(*-Text, +JSONTerm, +Options*) *[det]*

Convert between textual representation and a JSON term. In *write* mode, the option

as(*Type*)
defines the output type, which is one of `atom`, `string` or `codes`.

json_read(+*Stream, -Term*) *[det]*
json_read(+*Stream, -Term, +Options*) *[det]*
Read next JSON value from *Stream* into a Prolog term. The canonical representation for *Term* is:

- A JSON object is mapped to a term `json(NameValueList)`, where NameValueList is a list of Name=Value. Name is an atom created from the JSON string.
- A JSON array is mapped to a Prolog list of JSON values.
- A JSON string is mapped to a Prolog atom
- A JSON number is mapped to a Prolog number
- The JSON constants `true` and `false` are mapped -like JPL- to @(true) and @(false).
- The JSON constant `null` is mapped to the Prolog term @(null)

Here is a complete example in JSON and its corresponding Prolog term.

```
{ "name":"Demo term",
  "created": {
    "day":null,
    "month":"December",
    "year":2007
  },
  "confirmed":true,
  "members":[1,2,3]
}
```

```
json([ name='Demo term',
       created=json([day= @null, month='December', year=2007]),
       confirmed= @true,
       members=[1, 2, 3]
     ])
```

The following options are processed:

null(*NullTerm*)
Term used to represent JSON `null`. Default @(null)

true(*TrueTerm*)
Term used to represent JSON `true`. Default @(true)

false(*FalsTerm*)
Term used to represent JSON `false`. Default @(false)

value_string_as(*Type*)
Prolog type used for strings used as value. Default is `atom`. The alternative is `string`, producing a packed string object. Please note that `codes` or `chars` would produce ambiguous output and is therefore not supported.

If `json_read/3` encounters end-of-file before any real data it binds *Term* to the term @(end_of_file).

See also `json_read_dict/3` to read a JSON term using the version 7 extended data types.

json_write(+*Stream, +Term*) *[det]*
json_write(+*Stream, +Term, +Options*) *[det]*

Write a JSON term to *Stream*. The JSON object is of the same format as produced by `json_read/2`, though we allow for some more flexibility with regard to pairs in objects. All of Name=Value, Name-Value and Name(Value) produce the same output.

Values can be of the form #(*Term*), which causes *Term* to be *stringified* if it is not an atom or string. Stringification is based on `term_string/2`.

The version 7 *dict* type is supported as well. If the dicts has a *tag*, a property "type":"tag" is added to the object. This behaviour can be changed using the `tag` option (see below). For example:

```
?- json_write(current_output, point{x:1,y:2}).
{
  "type":"point",
  "x":1,
  "y":2
}
```

In addition to the options recognised by `json_read/3`, we process the following options are recognised:

width(+*Width*)
: *Width* in which we try to format the result. Too long lines switch from *horizontal* to *vertical* layout for better readability. If performance is critical and human readability is not an issue use *Width* = 0, which causes a single-line output.

step(+*Step*)
: Indentation increnment for next level. Default is 2.

tab(+*TabDistance*)
: Distance between tab-stops. If equal to Step, layout is generated with one tab per level.

serialize_unknown(+*Boolean*)
: If `true` (default `false`), serialize unknown terms and print them as a JSON string. The default raises a type error. Note that this option only makes sense if you can guarantee that the passed value is not an otherwise valid Prolog reporesentation of a Prolog term.

If a string is emitted, the sequence `</` is emitted as `<\/`. This is valid JSON syntax which ensures that JSON objects can be safely embedded into an HTML `<script>` element.

is_json_term(@*Term*) *[semidet]*
is_json_term(@*Term, +Options*) *[semidet]*

True if *Term* is a json term. *Options* are the same as for `json_read/2`, defining the Prolog representation for the JSON `true`, `false` and `null` constants.

json_read_dict(+*Stream, -Dict*) *[det]*
json_read_dict(+*Stream, -Dict, +Options*) *[det]*

Read a JSON object, returning objects as a dicts. The representation depends on the options, where the default is:

- String values are mapped to Prolog strings
- JSON `true`, `false` and `null` are represented using these Prolog atoms.
- JSON objects are mapped to dicts.
- By default, a `type` field in an object assigns a tag for the dict.

On addition to the options processed by `json_read/3`, `json_read_dict/3` processes this additional option:

tag(+*Name*)
When converting to/from a dict, map the indicated JSON attribute to the dict *tag*. No mapping is performed if *Name* is the empty atom (", default). See `json_read_dict/2` and `json_write_dict/2`.

json_write_dict(+*Stream, +Dict*) *[det]*
json_write_dict(+*Stream, +Dict, +Options*) *[det]*
Write a JSON term, represented using dicts. This is the same as `json_write/3`, but assuming the default representation of JSON objects as dicts.

atom_json_dict(+*Atom, -JSONDict, +Options*) *[det]*
atom_json_dict(-*Text, +JSONDict, +Options*) *[det]*
Convert between textual representation and a JSON term represented as a dict. *Options* are as for `json_read/3`. In *write* mode, the addtional option

as(*Type*)
defines the output type, which is one of `atom`, `string` or `codes`.

7.2 json_convert.pl: Convert between JSON terms and Prolog application terms

To be done
- Ignore extra fields. Using a partial list of *extra*?
- Consider a sensible default for handling JSON `null`. Conversion to Prolog could translate @null into a variable if the desired type is not `any`. Conversion to JSON could map variables to `null`, though this may be unsafe. If the Prolog term is known to be non-ground and JSON @null is a sensible mapping, we can also use this simple snipit to deal with that fact.

```
term_variables(Term, Vars),
maplist(=(@null), Vars).
```

The idea behind this module is to provide a flexible high-level mapping between Prolog terms as you would like to see them in your application and the standard representation of a JSON object as a Prolog term. For example, an X-Y point may be represented in JSON as `{"x":25, "y":50}`. Represented in Prolog this becomes `json([x=25,y=50])`, but this is a pretty non-natural representation from the Prolog point of view.

This module allows for defining records (just like `library(record)`) that provide transparent two-way transformation between the two representations.

```
:- json_object
       point(x:integer, y:integer).
```

This declaration causes `prolog_to_json/2` to translate the native Prolog representation into a JSON Term:

```
?- prolog_to_json(point(25,50), X).

X = json([x=25, y=50])
```

A `json_object/1` declaration can define multiple objects separated by a comma (,), similar to the `dynamic/1` directive. Optionally, a declaration can be qualified using a module. The conversion predicates `prolog_to_json/2` and `json_to_prolog/2` first try a conversion associated with the calling module. If not successful, they try conversions associated with the module `user`.

JSON objects have no *type*. This can be solved by adding an extra field to the JSON object, e.g. `{"type":"point", "x":25, "y":50}`. As Prolog records are typed by their functor we need some notation to handle this gracefully. This is achieved by adding +Fields to the declaration. I.e.

```
:- json_object
        point(x:integer, y:integer) + [type=point].
```

Using this declaration, the conversion becomes:

```
?- prolog_to_json(point(25,50), X).

X = json([x=25, y=50, type=point])
```

The predicate `json_to_prolog/2` is often used after `http_read_json/2` and `prolog_to_json/2` before `reply_json/1`. For now we consider them seperate predicates because the transformation may be too general, too slow or not needed for dedicated applications. Using a seperate step also simplifies debugging this rather complicated process.

current_json_object(*Term, Module, Fields*) *[multifile]*

Multifile predicate computed from the `json_object/1` declarations. *Term* is the most general Prolog term representing the object. *Module* is the module in which the object is defined and *Fields* is a list of `f(Name, Type, Default, Var)`, ordered by Name. Var is the corresponding variable in *Term*.

json_object +*Declaration*

Declare a JSON object. The declaration takes the same format as using in `record/1` from `library(record)`. E.g.

```
?- json_object
      point(x:int, y:int, z:int=0).
```

The type arguments are either types as know to `library(error)` or functor names of other JSON objects. The constant `any` indicates an untyped argument. If this is a JSON term, it becomes subject to `json_to_prolog/2`. I.e., using the type `list(any)` causes the conversion to be executed on each element of the list.

If a field has a default, the default is used if the field is not specified in the JSON object. Extending the record type definition, types can be of the form (Type1 | Type2). The type `null` means that the field may *not* be present.

Conversion of JSON to Prolog applies if all non-defaulted arguments can be found in the JSON object. If multiple rules match, the term with the highest arity gets preference.

prolog_to_json(*:Term, -JSONObject*) *[det]*

Translate a Prolog application *Term* into a JSON object term. This transformation is based on `:- json_object/1` declarations. If a `json_object/1` declaration declares a field of type `boolean`, commonly used thruth-values in Prolog are converted to JSON booleans. Boolean translation accepts one of `true`, `on`, `1`, @true, `false`, `fail`, `off` or `0`, @false.

Errors

- `type_error(json_term, X)`
- instantiation_error

json_to_prolog(*+JSON, -Term*) *[det]*

Translate a *JSON* term into an application term. This transformation is based on `:- json_object/1` declarations. An efficient transformation is non-trivial, but we rely on the assumption that, although the order of fields in *JSON* terms is irrelevant and can therefore vary a lot, practical applications will normally generate the *JSON* objects in a consistent order.

If a field in a json_object is declared of type `boolean`, @true and @false are translated to `true` or `false`, the most commonly used Prolog representation for truth-values.

7.3 http_json.pl: HTTP JSON Plugin module

See also

- JSON Requests are discussed in `http://json.org/JSONRequest.html`
- `json.pl` describes how JSON objects are represented in Prolog terms.
- `json_convert.pl` converts between more natural Prolog terms and json terms.

This module inserts the JSON parser for documents of MIME type `application/jsonrequest` and `application/json` requested through the `http_client.pl` library.

Typically JSON is used by Prolog HTTP servers. This module supports two JSON representations: the classical representation and the new representation supported by the SWI-Prolog version 7 extended data types. Below is a skeleton for handling a JSON request, answering in JSON using the classical interface.

```
handle(Request) :-
      http_read_json(Request, JSONIn),
      json_to_prolog(JSONIn, PrologIn),
      <compute>(PrologIn, PrologOut),                % application body
      prolog_to_json(PrologOut, JSONOut),
      reply_json(JSONOut).
```

When using dicts, the conversion step is generally not needed and the code becomes:

```
handle(Request) :-
      http_read_json_dict(Request, DictIn),
      <compute>(DictIn, DictOut),
      reply_json(DictOut).
```

This module also integrates JSON support into the http client provided by `http_client.pl`. Posting a JSON query and processing the JSON reply (or any other reply understood by `http_read_data/3`) is as simple as below, where Term is a JSON term as described in `json.pl` and reply is of the same format if the server replies with JSON.

```
      ...,
      http_post(URL, json(Term), Reply, [])
```

http_client:http_convert_data(*+In, +Fields, -Data, +Options*) *[multifile]*

Hook implementation that supports reading JSON documents. It processes the following option:

json_object *+ As*

Where *As* is one of `term` or `dict`. If the value is `dict`, `json_read_dict/3` is used.

json_type(*?MediaType*) *[semidet,multifile]*

True if *MediaType* is a JSON media type. `http_json:json_type/1` is a multifile predicate and may be extended to facilitate non-conforming clients.

Arguments

MediaType	is a term *Type/SubType*, where both *Type* and *SubType* are atoms.

http_client:post_data_hook(*+Data, +Out:stream, +HdrExtra*) *[semidet,multifile]*

Hook implementation that allows `http_post_data/3` posting JSON objects using one of the forms below.

```
http_post(URL, json(Term), Reply, Options)
http_post(URL, json(Term, Options), Reply, Options)
```

If Options are passed, these are handed to `json_write/3`. In addition, this option is processed:

json_object *As*

If *As* is `dict`, `json_write_dict/3` is used to write the output. This is default if `json(Dict)` is passed.

To be done avoid creation of intermediate data using chunked output.

http_read_json(*+Request, -JSON*) *[det]*

http_read_json(*+Request, -JSON, +Options*) *[det]*

Extract *JSON* data posted to this HTTP request. *Options* are passed to `json_read/3`. In addition, this option is processed:

json_object *+As*

One of `term` (default) to generate a classical Prolog term or `dict` to exploit the SWI-Prolog version 7 data type extensions. See `json_read_dict/3`.

Errors

- `domain_error(mimetype, Found)` if the mimetype is not known (see `json_type/1`).
- `domain_error(method, Method)` if the request is not a `POST` or `PUT` request.

http_read_json_dict(+*Request, -Dict*) *[det]*

http_read_json_dict(+*Request, -Dict, +Options*) *[det]*

Similar to `http_read_json/2`,3, but by default uses the version 7 extended datatypes.

reply_json(+*JSONTerm*) *[det]*

reply_json(+*JSONTerm, +Options*) *[det]*

Formulate a JSON HTTP reply. See `json_write/2` for details. The processed options are listed below. Remaining options are forwarded to `json_write/3`.

content_type(+*Type*)

The default `Content-type` is `application/json; charset=UTF8`. `charset=UTF8` should not be required because JSON is defined to be UTF-8 encoded, but some clients insist on it.

status(+*Code*)

The default status is 200. REST API functions may use other values from the 2XX range, such as 201 (created).

json_object *+As*

One of `term` (classical json representation) or `dict` to use the new dict representation. If omitted and Term is a dict, `dict` is assumed. SWI-Prolog Version 7.

reply_json_dict(+*JSONTerm*) *[det]*

reply_json_dict(+*JSONTerm, +Options*) *[det]*

As `reply_json/1` and `reply_json/2`, but assumes the new dict based data representation. Note that this is the default if the outer object is a dict. This predicate is needed to serialize a list of objects correctly and provides consistency with `http_read_json_dict/2` and friends.

8 MIME support

8.1 library(http/mimepack): Create a MIME message

Simple and partial implementation of MIME encoding. MIME is covered by RFC 2045. This library is used by e.g., `http_post_data/3` when using the form_data(+ListOfData) input specification.

MIME decoding is now arranged through library(mime) from the clib package, based on the external librfc2045 library. Most likely the functionality of this package will be moved to the same library someday. Packing however is a lot simpler then parsing.

mime_pack(+*Inputs, +Out:stream, ?Boundary*) *[det]*

Pack a number of inputs into a MIME package using a specified or generated boundary. The

generated boundary consists of the current time in milliseconds since the epoch and 10 random hexadecimal numbers. *Inputs* is a list of *documents* that is added to the mime message. Each element is one of:

Name **=** *Value*
: Name the document. This emits a header of the form below. The `filename` is present if Value is of the form file(File). Value may be any of remaining value specifications.

```
Content-Disposition: form-data; name="Name"[; filename="<File>"
```

html(*Tokens*)
: Tokens is a list of HTML tokens as produced by `html//1`. The token list is emitted using `print_html/1`.

file(*File*)
: Emit the contents of File. The `Content-type` is derived from the File using `file_mime_type/2`. If the content-type is `text/_`, the file data is copied in text mode, which implies that it is read in the default encoding of the system and written using the encoding of the *Out* stream. Otherwise the file data is copied binary.

stream(*In, Len*)
: Content is the next Len units from In. Data is copied using `copy_stream_data/3`. Units is bytes for binary streams and characters codes for text streams.

stream(*In*)
: Content of the stream In, copied using `copy_stream_data/2`. This is often used with memory files (see `new_memory_file/1`).

mime(*Attributes, Value,* **)**
: Create a MIME header from Attributes and add Value, which can be any of remaining values of this list. Attributes may contain type(ContentType) and/or character_set(CharSet). This can be used to give a content-type to values that otherwise do not have a content-type. For example:

```
mime([type(text/html)], '<b>Hello World</b>', [])
```

mime(*, , Parts*)
: Creates a nested multipart MIME message. Parts is passed as *Inputs* to a recursive call to `mime_pack/2`.

Atomic
: Atomic values are passed to `write/1`. This embeds simple atoms and numbers.

Parameters

Out	is a stream opened for writing. Typically, it should be opened in text mode using UTF-8 encoding.

bug Does not validate that the boundary does not appear in any of the input documents.

9 Security

Writing servers is an inherently dangerous job that should be carried out with some considerations. You have basically started a program on a public terminal and invited strangers to use it. When using

the interactive server or inetd based server the server runs under your privileges. Using CGI scripted it runs with the privileges of your web-server. Though it should not be possible to fatally compromise a Unix machine using user privileges, getting unconstrained access to the system is highly undesirable.

Symbolic languages have an additional handicap in their inherent possibilities to modify the running program and dynamically create goals (this also applies to the popular Perl and PHP scripting languages). Here are some guidelines.

- *Check your input*
  Hardly anything can go wrong if you check the validity of query-arguments before formulating an answer.

- *Check filenames*
  If part of the query consists of filenames or directories, check them. This also applies to files you only read. Passing names as `/etc/passwd`, but also `../../../../../etc/passwd` are tried by hackers to learn about the system they want to attack. So, expand provided names using `absolute_file_name/[2,3]` and verify they are inside a folder reserved for the server. Avoid symbolic links from this subtree to the outside world. The example below checks validity of filenames. The first call ensures proper canonisation of the paths to avoid an mismatch due to symbolic links or other filesystem ambiguities.

```
check_file(File) :-
        absolute_file_name('/path/to/reserved/area', Reserved),
        absolute_file_name(File, Tried),
        sub_atom(Tried, 0, _, _, Reserved).
```

- *Check scripts*
  Should input in any way activate external scripts using `shell/1` or `open(pipe(Command), ...)`, verify the argument once more. Use `process_create/3` in preference over `shell/1` as this function avoids stringification of arguments (Unix) or ensures proper quoting of arguments (Windows).

- *Check meta-calling*
  The attractive situation for you and your attacker is below:

```
reply(Query) :-
        member(search(Args), Query),
        member(action=Action, Query),
        member(arg=Arg, Query),
        call(Action, Arg).              % NEVER EVER DO THIS!
```

  All your attacker has to do is specify *Action* as `shell` and *Arg* as `/bin/sh` and he has an uncontrolled shell!

10 Tips and tricks

- *URL Locations*
  With an application in mind, it is tempting to make all URL locations short and directly con-

nected to the root (/). This is *not* a good idea. It is adviced to have all locations in a server below a directory with an informative name. Consider to make the root location something that can be changed using a global setting.

  - Page generating code can easily be reused. Using locations directly below the root however increases the likelihood of conflicts.
  - Multiple servers can be placed behind the same public server as explained in section 3.12.7. Using a common and fairly unique root, redirection is much easier and less likely to lead to conflicts.

- *Debugging*
  Debugging multi-threaded applications is possible using the graphical debugger. This implies requires that the xpce extension package must be installed. Spy-points may be placed using `tspy/1`.

11 Status

The SWI-Prolog HTTP library is in active use in a large number of projects. It is considered one of the SWI-Prolog core libraries that is actively maintained and regularly extended with new features. This is particularly true for the multi-threaded server. The inetd based server may be applicable for infrequent requests where the startup time is less relevant. The XPCE based server is considered obsolete.

This library is by no means complete and you are free to extend it.

Index