

SWI-Prolog HTTP support

Jan Wielemaker
HCS,
University of Amsterdam
The Netherlands
E-mail: `J.Wielemaker@uva.nl`

December 2, 2008

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.

Contents

1	Introduction	4
2	The HTTP client libraries	4
2.1	The <code>http/http_open</code> library	4
2.2	The <code>http/http_client</code> library	6
2.2.1	The MIME client plug-in	9
2.2.2	The SGML client plug-in	9
3	The HTTP server libraries	9
3.1	The ‘Body’	10
3.1.1	Returning special status codes	11
3.2	Dispatching HTTP locations over predicates	11
3.3	HTTP Session management	13
3.4	HTTP Authentication	14
3.5	Get parameters from HTML forms	15
3.6	Request format	17
3.6.1	Handling POST requests	18
3.7	Running the server	19
3.7.1	Common server interface options	19
3.7.2	Multi-threaded Prolog	19
3.7.3	From an interactive Prolog session using XPCE	22
3.7.4	From (Unix) <code>inetd</code>	22
3.7.5	MS-Windows	23
3.7.6	As CGI script	23
3.7.7	Using a reverse proxy	23
3.8	The wrapper library	24
3.9	<code>http_log.pl</code> – HTTP Logging module	25
3.10	Debugging Servers	26
3.11	Handling HTTP headers	26
3.12	The <code>http/html_write</code> library	27
3.12.1	Emitting HTML documents	30
3.12.2	Repositioning HTML for CSS and javascript links	30
3.12.3	Adding rules for <code>html//1</code>	32
3.12.4	Generating layout	32
3.12.5	Examples	33
3.12.6	Remarks on the <code>http/html_write</code> library	34
3.13	<code>http_path.pl</code> – Abstract specification of HTTP server locations	34
3.14	Security	35
3.15	Tips and tricks	36
4	Transfer encodings	36
4.1	The <code>http/http_chunked</code> library	37

5	Supporting JSON	37
5.1	json.pl – Reading and writing JSON serialization	37
5.2	json_convert.pl – Convert between JSON terms and Prolog application terms	39
5.3	http_json.pl – HTTP JSON Plugin module	41
6	Status	41

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. It is the basis of popular agent protocols such as SOAP and FIPA.

In this document we describe a modular infra-structure to access web-servers from SWI-Prolog and turn Prolog into a web-server. The server code is designed to allow the same ‘body’ to be used from an interactive server for debugging or providing services from otherwise interactive applications, run the body from an *inetd* super-server or as a CGI script behind a generic web-server.

The design of this module is different from the competing XPCE-based HTTP server located in `http/httpd.pl`, which intensively uses XPCE functionality to reach its goals. Using XPCE is not very suitable for CGI or *inetd*-driven servers due to required X11 connection and much larger footprint.

Acknowledgements

This work has been carried out under the following projects: GARP, MIA, IBROW and KITS. The following people have pioneered parts of this library and contributed with bug-report and suggestions for improvements: Anjo Anjewierden, Bert Bredeweg, Wouter Jansweijer and Bob Wielinga.

2 The HTTP client libraries

This package provides two packages for building HTTP clients. The first, `http/http_open` is a very 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 The `http/http_open` library

The library `http/http_open` provides a very simple mechanism to read data from an HTTP server using the HTTP 1.0 protocol and HTTP GET access method. It defines one predicate:

`http_open(+URL, -Stream, +Options)`

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 compatible to `parse_url/2`. 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:

`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 unified with the canonical version of *URL* using

```
parse_url(URL, Parts),
parse_url(FinalURL, 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 ("").

method(Method)

One of `get` (default) or `head`. 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.

proxy(+Host, +Port)

Use an HTTP proxy to connect to the outside world.

authorization(+Authorization)

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

request_header(+Name = +Value)

Additional name-value parts are added in the order of appearance to the HTTP request header. No interpretation is done.

size(-Size)

If provided *Size* is unified with the value of the `Content-Length` fields of the reply-header.

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`).

user_agent(+Agent)

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

Here is a simple example:

```
?- http_open('http://www.swi-prolog.org/news.html', In, []),
   copy_stream_data(In, user_output),
   close(In).
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">

<HTML>
<HEAD>
<TITLE>News</TITLE>
</HEAD>
...
```

The example below fetches the modification time of a web-page. Note that *Modified* is '' 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).
close(In).

```

The `http_open/3` predicate is designed to be lightweight. The `http_get/3` and `http_post/4` predicates provide more powerful, but also more complicated and resource-intensive alternatives to `http_open/3`. The `http_open/3` predicate supports transfer-encoding hooks as described in section 4. In particular, loading `http/http_chunked` provides support for HTTP 1.1 chunked encoding transfer.

http_set_authorization(+URLPrefix, +Authorization)

Set user/password to supply with URLs that start with *URLPrefix*. If *Authorization* is the atom `=====`, possibly defined authorization is cleared. For example:

```

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

```

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.

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*).

user_agent(+Agent)

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

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, XPC object, 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

codes

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

content_type(Type)

Override 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.

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

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 [3.12](#).

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`.

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.

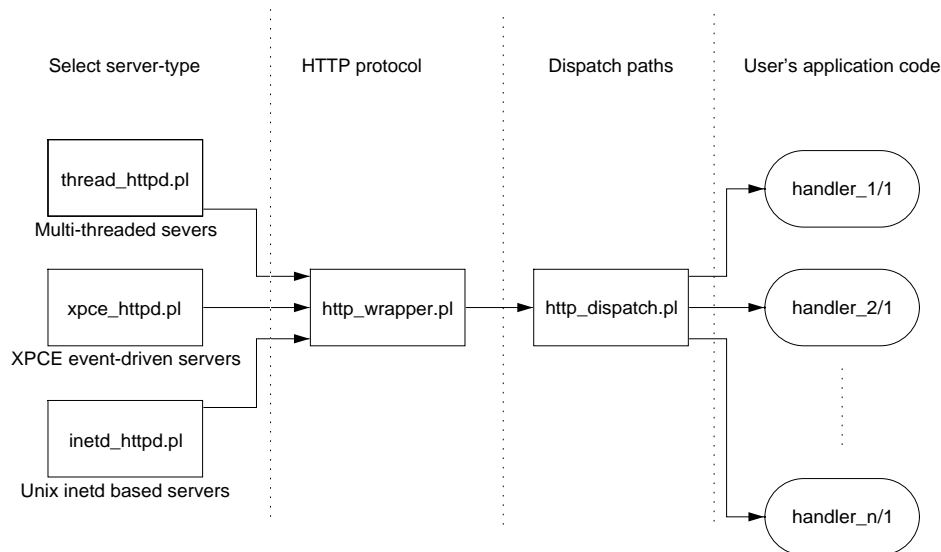


Figure 1: Design of the HTTP server

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 3.6) 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 `Transfer-encoding: chunked`, causing it to use chunked encoding if the client allows for it. See also section 4 and the `chunked` option in `http_handler/3`. Other header lines are passed verbatim to the client. Typical examples are `Set-Cookie` and authentication headers (see section 3.4).

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.

3.2 Dispatching HTTP locations over predicates

The library `http/http_dispatch.pl` can be between `http_wrapper/5` and the user handlers as described in section 3.1 to distribute locations over multiple predicates. This has several advantages:

- It makes a distinction between failure of the predicate and undefined locations, giving a server error on failure and a 404 existence error on undefined locations.
- It is much easier to put a spy-point on specific locations or find the code implementing a location. In addition, the `edit/1` interface is extended to deal with locations. This allows for `?- edit('/path1/path2') .` to open an editor on the implementation of the HTTP location `/path1/path2`.

We introduce the library with a small but complete example combining the core HTTP libraries. To test the example on port 5000, load the code, run `?- server(5000) .` and direct your browser to `http://localhost:5000`.

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

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

:- http_handler('/', root, []).
:- http_handler('/hello/world', hello_world, []).

root(_Request) :-
    reply_html_page([ title('Demo server')
                      ],
                    [ p(a(href('hello/world'), hello))
                      ]).
```

```
hello_world(_Request) :-
    reply_html_page([ title('Hello World')
                      ],
                    [ h1('Hello World'),
                      p('This is my first page')
                    ]).
```

http_dispatch(+Request)

Dispatch *Request* to a predicate defined to handle the path component (location) of the request. Locations are declared using `http_handler/3`. The predicate `http_dispatch/1` is normally called from `http_server/2`.

http_handler(+Location, :Closure, +Options)

Call `call(Closure, Request)` if an HTTP request for the path *Location* is dispatched by `http_dispatch/1`. Options include:

authentication(+Type)

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

chunked

Enable chunked transfer-encoding if the client allows for it. See section 4.

id(+ID)

Identifier of the handler. The default identifier is the predicate name. Used by `http_location_by_id/2`.

time_limit(+Spec)

Specify a (wall-)time limit to handle the request. *Spec* is one of `infinite`, `default` or a number expressing seconds. If `default` or omitted, the setting (see `setting/4`) `http:time_limit` is used.

prefix

If present, call handler for all paths that *start with* the given path. If multiple specifications match the most specific one is used.

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.

spawn(+Spec)

Handle the request on a separate thread. *Spec* is either the name of a thread pool (see `thread_pool_create/3` from `library(thread_pool)`) or a set of options that are passed to `thread_create/3`. See also `http_spawn/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.

http_delete_handler(+Path)

Delete handler for *Path*. Typically, this should only be used for handlers that are registered dynamically. Use the priority option to overrule an existing handler.

http_location_by_id(+ID, -Location)

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.
- `<module>.<predicate>`
The module-qualified name of the predicate.
- `<predicate>`

http_current_handler(?Location, ?Closure)

True if *Location* is handled by *Closure*. If *Location* is given, the *Closure* with highest priority is returned.

http_reply_file(+FileSpec, +Options, +Request)

Reply a file. The argument order is defined to allow using the predicate as a closure for `http_handler/3`. Options include:

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/`

Below is an example. Note that the file argument can be a specification for `absolute_file_name/3` that is searched over application libraries defined using `file_search_path/2`.

```
:- http_handler('/css/myapp.css',
               http_reply_file('myapp.css', []),
               []).
```

3.3 HTTP Session management

The library `http/http_session.pl` provides cookie-based session management. The library installs a session-id cookie using the hook `http:request_expansion/2`. It allows querying the session and provides a simple assert/retract based store to store information related to a session. Note that session management only works with the threaded and XPCE based server frameworks as the `inetd` based server starts a server for each request.

The examples contain the file `calc.pl`, which realises a simple calculator with internal state.

http.set_session_options(+Options)

Set options for the session manager. Defined options are:

timeout(+Seconds)

Max idle time of a session. Session cookies are deleted if no request is received within the specified time. The value '0' disables timeout handling.

cookie(+Atom)

Name of the cookie to use for session management. The default is `swipl_session`.

path(+Atom)

Path with which to associate the session management. Default is `/`, associating it with the entire server.

http_session_id(-Id)

Returns an identifier for the current session. The identifier is an atom.

http_current_session(?Id, ?Data)

Enumerate sessions and associated data. All sessions have the *Data* item `idle(Seconds)`, describing the current idle-time of the session. Other data elements are added by the user using `http_session_assert/1` and friends.

http_session_asserta(+Term)

Associate *Term* with the current session, before any other associated term.

http_session_assert(+Term)

Associate *Term* with the current session, after any other associated term.

http_session_retract(?Term)

Non-deterministically retract terms associated with the current session.

http_session_retractall(+Term)

Retract all matching terms from associated with the current session.

http_session_data(?Term)

Enumerate all associated terms that unify with *Term*.

3.4 HTTP Authentication

The module `http/http_authenticate` provides the basics to validate an HTTP Authorization error. User and password information are read from a Unix/Apache compatible password file. This information, as well as the validation process is cached to achieve optimal performance.

http_authenticate(T)

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

basic(+PasswordFile)

Use HTTP Basic authentication 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, User)  
-> true  
; throw(http_reply(authorise(basic, Realm)))  
).
```

Alternatively `basic(+PasswordFile)` can be passed as an option to `http_handler/3`.

3.5 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 3.6.

`http_parameters(+Request, ?Parameters)`

The predicate 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(form_data, Name), _)` is thrown. Defined options are:

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.

zero_or_more

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.

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.

number

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

integer

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

float

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

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.

Below is an example

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

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.

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]).

```

3.6 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.

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 canonised 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, 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)
    ].
```

3.6.1 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/httpmime-plugin` is loaded.

3.7 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 PIDoc 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 3.7.7.

- *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 `xpce_httpd` for an event-driven server*

This approach provides a single-threaded event-driven application. The clients talk to XPCE sockets that collect an HTTP request. The server infra-structure can talk to multiple clients simultaneously, but once a request is complete the wrappers call the user’s goal and blocks all further activity until the request is handled. Requests from multiple clients are thus fully serialised in one Prolog process.

This server setup is very suitable for debugging as well as embedded server in simple applications in a fairly controlled environment.

- *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.

3.7.1 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.

3.7.2 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 other (XPCE and inetd) servers, 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(?Port)

Port the server should listen to. If unbound *Port* is unified with the selected free port.

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 plain 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.current_server(?:Goal, ?Port)

Query the running servers. Note that `http_server/3` can be called multiple times to create multiple servers on different ports.

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.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` if 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 recommend 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)]).

```

3.7.3 From an interactive Prolog session using XPCE

The `http/xpce_httpd.pl` provides the infrastructure to manage multiple clients with an event-driven control-structure. This version can be started from an interactive Prolog session, providing a comfortable infra-structure to debug the body of your server. It also allows the combination of an (XPCE-based) GUI with web-technology in one application.

http_server(:Goal, +Options)

Create an instance of *interactive_httpd*. *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. Currently no options are defined.

The file `demo_xpce` gives a typical example of this wrapper, assuming `demo_body` defines the predicate `reply/1`.

```

:- use_module(xpce_httpd).
:- use_module(demo_body).

server(Port) :-
    http_server(reply, Port, []).

```

The created server opens a server socket at the selected address and waits for incoming connections. On each accepted connection it collects input until an HTTP request is complete. Then it opens an input stream on the collected data and using the output stream directed to the XPCE *socket* it calls `http_wrapper/5`. This approach is fundamentally different compared to the other approaches:

- *Server can handle multiple connections*
When *inetd* will start a server for each *client*, and CGI starts a server for each *request*, this approach starts a single server handling multiple clients.
- *Requests are serialised*
All calls to *Goal* are fully serialised, processing on behalf of a new client can only start after all previous requests are answered. This is easier and quite acceptable if the server is mostly inactive and requests take not very long to process.
- *Lifetime of the server*
The server lives as long as Prolog runs.

3.7.4 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
```

3.7.5 MS-Windows

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

3.7.6 As CGI script

To be done.

3.7.7 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.

3.8 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 3.3) 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*.

3.9 http_log.pl – 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 be 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]

Returns handle to open logfile. Fails if no logfile is open and none is defined.

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.

3.10 Debugging Servers

The library `http/http_error.pl` 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.

3.11 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 3.12.

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`).

3.12 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 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 `sformat/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.
- *\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>;`. As Prolog understands Unicode and automatically inserts appropriate entity declarations, this is normally not needed.

- `Tag(Content)`
Emit HTML element *Tag* using *Content* and no attributes. *Content* is handed to `html//1`. See section 3.12.4 for details on the automatically generated layout.
- `Tag(Attributes, Content)`
Emit HTML element *Tag* using *Attributes* and *Content*. *Attributes* is either a single attribute or a list of attributes. Each attribute 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*
 - `encode(Atom)`
Use `www_form_encode/2` to create a valid URL component.
 - `location_by_id(ID)`
HTTP location of the HTTP handler with given ID. See `http_location_by_id/2`.
 - *List*
A 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.

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.

3.12.1 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(I)` 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*)

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 ...')
                    ]).
```

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.

3.12.2 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 `mainman/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 channel 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 `xhtml_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.

3.12.3 Adding rules for html//1

In some cases it is practical to extend the translations imposed by `html//1`. When using XPCE for example, it is comfortable to be able defining default translation to HTML for objects. We also 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 `<&>`.

3.12.4 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.

3.12.5 Examples

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)
    ).

```

3.12.6 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 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).

The approach presented in this library has been used in combination with `http/httpd` in three projects: viewing RDF in a browser, selecting fragments from an analysed document and presenting parts of the XPCE documentation using a browser. It has proven to be able to deal with generating pages quickly and comfortably.

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.

3.13 `http_path.pl` – Abstract specification of HTTP server locations

To be done

- Make this module replace the `http:prefix` option.
- Remove hard-wired support for `prefix()`.

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 specification 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.

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_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.

3.14 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 java 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 experienced 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),
    atom_concat(Reserved, _, Tried).

```

- *Check scripts*

Should input in any way activate external scripts using `shell/1` or `open(pipe(Command), ...)`, verify the argument once more.

- *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!

3.15 Tips and tricks

- *URL Locations*

With an application in mind, it is tempting to make all URL locations short and directly connected to the root (`/`). This is *not* a good idea. It is advised 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 3.7.7. Using a common and fairly unique root, redirection is much easier and less likely to lead to conflicts.

- *Debugging*

Please check the section “Thread-support library(threadutil)” of the SWI-Prolog reference manual.

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://gollem.science.uva.nl/SWI-Prolog/pldoc/> for details.

5 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.

5.1 `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. The canonical Prolog representation for a JSON value is defined as:

- 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]
      ])
```

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. *Options* are:

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.

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. 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 increment for next level. Default is 2.

tab(+TabDistance)

Distance between tab-stops. If equal to `Step`, layout is generated with one tab per level.

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.

5.2 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 snippet 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 separate predicates because the transformation may be too general, too slow or not needed for dedicated applications. Using a separate step also simplifies debugging this rather complicated process.

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).
```

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 truth-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.

5.3 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. Below is a skeleton for handling a JSON request, answering in JSON.

```
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).
```

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_read_json(+Request, -JSON) [det]

http_read_json(+Request, -JSON, +Options) [det]

Extract *JSON* data posted to this HTTP request.

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 request.

reply_json(+JSONTerm) [det]

reply_json(+JSONTerm, +Options) [det]

Formulate a JSON HTTP reply. See `json_write/2` for details. *Options* accepts `content_type(+Type)` and options accepted by `json_write/3`.

6 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 XPC and inetd based servers are not widely used.

This library is by no means complete and you are free to extend it.

Index

absolute_file_name/3, 13
absolute_file_name/[2
3], 35
atom_json_term/3, 38

chunked
 encoding, 36
close/1, 4
crypt/2, 14

deflate
 encoding, 36

edit/1, 11

file_mime_type/2, 8
file_search_path/2, 13
format/2, 27, 34

goal_expansion/2, 34

html/1, 27
html_begin/1, 29
html_end/1, 29
html_post//2, 30
html_print/[1
2], 30
html_print_length/2, 30
html_quoted/1, 32
html_quoted_attribute/1, 32
html_receive//1, 31
html_receive//2, 31
html_write *library*, 8
html_write:expand/1, 32
html_write:layout/3, 32
http/html_write *library*, 26, 27, 34
http/http_chunked *library*, 6, 37
http/http_chunked.pl *library*, 37
http/http_client *library*, 4, 6
http/http_error.pl *library*, 26
http/http_header *library*, 17, 26
http/http_mime_plugin *library*, 9, 18
http/http_open *library*, 4
http/http_open.pl *library*, 37
http/http_parameters *library*, 15
http/http_session.pl *library*, 13

http/http_sgml_plugin *library*, 9
http/http_wrapper.pl *library*, 24
http/httpd *library*, 34
http/httpd.pl *library*, 4
http/thread_httpd.pl *library*, 19
http/xpce_httpd.pl *library*, 22
http:request_expansion/2, 25
http_absolute_location/3, 35
http_authenticate/+Type
 +Request
 -User, 14
http_authenticate/3, 15
http_chunked_open/3, 37
http_current_handler/2, 13
http_current_request/1, 25
http_current_server/2, 21
http_current_session/2, 14
http_current_worker/2, 21
http_delete_handler/1, 13
http_dispatch/1, 12
http_get/3, 6, 7
http_handler/3, 10, 12, 13, 15, 21, 28
http_location_by_id/2, 12, 13, 28
http_log/2, 25
http_log_stream/1, 25
http_open/3, 4, 6, 37
http_parameters/2, 15, 16
http_parameters/3, 16
http_post/4, 6, 7
http_post_data/3, 7, 8
http_read_data/3, 6, 7, 18
http_read_json/2, 41
http_read_json/3, 41
http_read_request/2, 7, 17, 18
http_relative_path/2, 25
http_reply/3, 11, 26
http_reply_file/3, 13
http_server/1, 23
http_server/2, 12, 22, 23
http_server/3, 20, 21
http_session_assert/1, 14
http_session_asserta/1, 14
http_session_data/1, 14
http_session_id/1, 14

http_session_retract/1, 14
http_session_retractall/1, 14
http_set_authorization/2, 5, 6
http_set_session_options/1, 14
http_spawn/2, 12, 21
http_stop_server/2, 21
http_workers/2, 20, 21
http_wrapper *library*, 30
http_wrapper/5, 10, 11, 15, 22–25

inetd_httpd *library*, 19
interactive_httpd *class*, 22
is_json_term/1, 39
is_json_term/2, 39

json_object/1, 40
json_read/2, 38
json_read/3, 38
json_to_prolog/2, 40
json_write/2, 39
json_write/3, 39

load_structure/3, 9

make/0, 12
mime_pack/3, 9

page/1, 29
page/2, 28
page/[1
2], 29
parse_time/2, 5
parse_url/2, 4
pp/1, 18
predicate/5, 33
print_html/1, 26, 30
print_html/2, 30
print_html/[1
2], 27, 32
print_message/2, 27
prolog_to_json/2, 40

reply/1, 22
reply_html_page/2, 30
reply_json/1, 41
reply_json/2, 41

set_lang/1, 28
set_stream/2, 5, 6

setting/2, 13
setting/4, 12
sformat/3, 27
sgml *library*, 9, 32
shell/1, 36
socket *class*, 22
ssl *library*, 20
ssl_init/3, 20

tcp_accept/3, 25
thread_create/3, 12, 21
thread_create_in_pool/4, 21
thread_httpd *library*, 19
thread_pool_create/3, 12, 21
throw/1, 11
tspy/1, 19

www_form_encode/2, 28

xhtml_ns/2, 32
xpce_httpd *library*, 19

zlib.pl *library*, 37