Hunchentoot - The Common Lisp web server formerly known as TBNL

Abstract

Hunchentoot is a web server written in Common Lisp and at the same time a toolkit for building dynamic websites. As a stand-alone web server, Hunchentoot is capable of HTTP/1.1 chunking (both directions), persistent connections (keep-alive), and SSL.

Hunchentoot provides facilities like automatic session handling (with and without cookies), logging, customizable error handling, and easy access to GET and POST parameters sent by the client. It does *not* include functionality to programmatically generate HTML output. For this task you can use any library you like, e.g. (shameless self-plug) CL-WHO or HTML-TEMPLATE.

Hunchentoot talks with its front-end or with the client over TCP/IP sockets and optionally uses multiprocessing to handle several requests at the same time. Therefore, it cannot be implemented completely in portable Common Lisp. It currently works with LispWorks and all Lisps which are supported by the compatibility layers usocket and Bordeaux Threads.

Hunchentoot comes with a BSD-style license so you can basically do with it whatever you want.

Hunchentoot is (or was) for example used by QuickHoney, City Farming, Heike Stephan.

Download shortcut: http://weitz.de/files/hunchentoot.tar.gz.

Contents

1. Abstract
2. Contents
3. Download and installation
    1. Running Hunchentoot on port 80
    2. Hunchentoot behind a proxy
4. Support
5. Your own webserver (the easy teen-age New York version)
6. Third party documentation and add-ons
7. Function and variable reference
    1. Acceptors
    2. Customizing acceptor behaviour
    3. An example of how to subclass ACCEPTOR
    4. Taskmasters
    5. Request dispatch and handling
    6. Using the easy-handler framework
    7. Request objects
    8. Reply objects
    9. Sessions
    10. Customizing session behaviour
    11. Cookies
    12. Logging
    13. Conditions and error handling
    14. Miscellaneous
8. Testing
9. Debugging
10. History
11. Symbol index
12. Acknowledgements

Download and installation

Hunchentoot depends on a couple of other Lisp libraries which you'll need to install first:

- Pierre R. Mai's MD5,
- Kevin Rosenberg's CL-BASE64,
- Janis Dzerins' RFC2388,
- Peter Seibel's CL-FAD,
- Gary King's trivial-backtrace,
- Erik Huelsmann's usocket (unless you're using LispWorks),
- Greg Pfeil's Bordeaux Threads (unless you're using LispWorks),
- David Lichteblau's CL+SSL (unless you're using LispWorks),
- and my own FLEXI-STREAMS (0.12.0 or higher), Chunga (1.0.0 or higher), and CL-PPCRE (plus CL-WHO for the example code and Drakma for the tests).

Make sure to use the *newest* versions of all of these libraries (which might themselves depend on other libraries) - try the repository versions if you're in doubt. Note: You can compile Hunchentoot without SSL support - and thus without the need to have CL+SSL - if you add :HUNCHENTOOT-NO-SSL to *FEATURES* *before* you compile it.

Hunchentoot will only work with Lisps where the character codes of all Latin-1 characters coincide with their Unicode code points (which is the case for all current implementations I know).

Hunchentoot itself together with this documentation can be downloaded from https://github.com/edicl/hunchentoot/archive/v1.2.38.tar.gz. The current version is 1.2.38.

The preferred method to compile and load Hunchentoot is via ASDF. If you want to avoid downloading and installing all the dependencies manually, give Zach Beane's excellent Quicklisp system a try.

Hunchentoot and its dependencies can also be installed with clbuild. There's also a port for Gentoo Linux thanks to Matthew Kennedy.

The current development version of Hunchentoot can be found at https://github.com/edicl/hunchentoot. If you want to send patches, please fork the github repository and send pull requests.

Running Hunchentoot on port 80

Hunchentoot does not come with code to help with running it on a privileged port (i.e. port 80 or 443) on Unix-like operating systems. Modern Unix-like systems have specific, non-portable ways to allow non-root users to listen to privileged ports, so including such functionality in Hunchentoot was considered unnecessary. Please refer to online resources for help. At the time of this writing, the YAWS documentation has a comprehensive writeup on the topic.

Hunchentoot behind a proxy

If you're feeling unsecure about exposing Hunchentoot to the wild, wild Internet or if your Lisp web application is part of a larger website, you can hide it behind a proxy server. One approach that I have used several times is to employ Apache's mod_proxy module with a configuration that looks like this:

```
ProxyPass /hunchentoot http://127.0.0.1:3000/hunchentoot
ProxyPassReverse /hunchentoot http://127.0.0.1:3000/hunchentoot
```

This will tunnel all requests where the URI path begins with "/hunchentoot" to a (Hunchentoot) server listening on port 3000 on the same machine.

Of course, there are several other (more lightweight) web proxies that you could use instead of Apache.

Support

The development version of Hunchentoot can be found on github. Please use the github issue tracking system to submit bug reports. Patches are welcome, please use GitHub pull requests. If you want to make a change, please read this first.

Your own webserver (the easy teen-age New York version)

Starting your own web server is pretty easy. Do something like this:

```
(hunchentoot:start (make-instance 'hunchentoot:easy-acceptor :port 4242))
```

That's it. Now you should be able to enter the address "http://127.0.0.1:4242/" in your browser and see something, albeit nothing very interesting for now.

By default, Hunchentoot serves the files from the *www/* directory in its source tree. In the distribution, that directory contains a HTML version of the documentation as well as the error templates. The location of the document root directory can be specified when creating a new ACCEPTOR instance by the way of the ACCEPTOR-DOCUMENT-ROOT. Likewise, the location of the error template directory can be specified by the ACCEPTOR-ERROR-TEMPLATE-DIRECTORY. Both ACCEPTOR-DOCUMENT-ROOT and ACCEPTOR-ERROR-TEMPLATE-DIRECTORY can be specified using a logical pathname, which will be translated once when the ACCEPTOR is instantiated.

The EASY-ACCEPTOR class implements a framework for developing web applications. Handlers are defined using the DEFINE-EASY-HANDLER macro. Request dispatching is performed according to the list of dispatch functions in *DISPATCH-TABLE*. Each of the functions on that list is called to determine whether it wants to handle the request, provided as single argument. If a dispatcher function wants to handle the request, it returns another function to actually create the desired page.

DEFINE-EASY-HANDLER is accompanied by a set of dispatcher creation functions that can be used to create dispatchers for standard tasks. These are documented in the subchapter on easy handlers

Now be a bit more adventurous, try this

```
(hunchentoot:define-easy-handler (say-yo :uri "/yo") (name)
  (setf (hunchentoot:content-type*) "text/plain")
  (format nil "Hey~@[ ~A~]!" name))
```

and see what happens at "http://127.0.0.1:4242/yo" or "http://127.0.0.1:4242/yo?name=Dude" .

Hunchentoot comes with a little example website which you can use to see if it works and which should also demonstrate a couple of the things you can do with Hunchentoot. To start the example website, enter the following code into your listener:

```
(asdf:oos 'asdf:load-op :hunchentoot-test)
```

Now go to "http://127.0.0.1:4242/hunchentoot/test" and play a bit.

Third party documentation and add-ons

Adam Petersen has written a book called "Lisp for the Web" which explains how Hunchentoot and some other libraries can be used to build web sites.

Here is some software which extends Hunchentoot or is based on it:

- Clack is a web server abstraction layer, defaulting to Hunchentoot.
- hunchentoot-cgi (by Cyrus Harmon) provides CGI handlers for Hunchentoot.
- CL-WEBDAV is a WebDAV server based on Hunchentoot.
- RESTAS is a web framework based on Hunchentoot. Caveman, Radiance, Snooze or again Weblocks are frameworks compatible with it.

Function and variable reference

Acceptors

If you want Hunchentoot to actually do something, you have to create and start an acceptor. You can also run several acceptors in one image, each one listening on a different different port.

[Standard class]
acceptor

> To create a Hunchentoot webserver, you make an instance of this class or one of its subclasses and use the generic function START to start it (and STOP to stop it). Use the :port initarg if you don't want to listen on the default http port 80. If 0 is specified for the port, the system chooses a random port to listen on. The port number choosen can be retrieved using the ACCEPTOR-PORT accessor. The port number chosen is retained across stopping and starting the acceptor.
>
> There are other initargs most of which you probably won't need very often. They are explained in detail in the docstrings of the slot definitions.
>
> Unless you are in a Lisp without MP capabilities, you can have several active instances of ACCEPTOR (listening on different ports) at the same time.

[Standard class]
ssl-acceptor

> Create and START an instance of this class (instead of ACCEPTOR) if you want an https server. There are two required initargs, :SSL-CERTIFICATE-FILE and :SSL-PRIVATEKEY-FILE, for pathname designators denoting the certificate file and the key file in PEM format. On LispWorks, you can have both in one file in which case the second initarg is optional. You can also use the :SSL-PRIVATEKEY-PASSWORD initarg to provide a password (as a string) for the key file (or NIL, the default, for no password).
>
> The default port for SSL-ACCEPTOR instances is 443 instead of 80

[Generic function]
start *acceptor => acceptor*

> Starts *acceptor* so that it begins accepting connections. Returns the acceptor.

[Generic function]
stop *acceptor &key soft => acceptor*

> Stops the *acceptor* so that it no longer accepts requests. If *soft* is true, and there are any requests in progress, wait until all requests are fully processed, but meanwhile do not accept new requests. Note that *soft* must not be set when calling stop from within a request handler, as that will deadlock.

[Generic function]
started-p *acceptor => generalized-boolean*

> Tells if *acceptor* has been started. The default implementation simply queries *acceptor* for its listening status, so if T is returned to the calling thread, then some thread has called start or some thread's call to stop hasn't finished. If NIL is returned either some thread has called stop, or some thread's call to start hasn't finished or start was never called at all for *acceptor*.

[Special variable]
acceptor

> The current ACCEPTOR object in the context of a request.

[Generic function]
acceptor-listen-backlog *listen-backlog => number-of-pending-connections*

Number of pending connections allowed in the listen socket before the kernel rejects further incoming connections. Non-LispWorks only.

[Generic readers]
acceptor-address *acceptor => address*
acceptor-port *acceptor => port*
acceptor-read-timeout *acceptor => read-timeout*
acceptor-ssl-certificate-file *ssl-acceptor => ssl-certificate-file*
acceptor-ssl-privatekey-file *ssl-acceptor => ssl-privatekey-file*
acceptor-ssl-privatekey-password *ssl-acceptor => ssl-privatekey-password*
acceptor-write-timeout *acceptor => write-timeout*

These are readers for various slots of ACCEPTOR objects (and some of them obviously only make sense for SSL-ACCEPTOR objects). See the docstrings of these slots for more information and note that there are corresponding initargs for all of them.

[Generic accessors]
acceptor-access-log-destination *acceptor => (or pathname null)*
(setf (**acceptor-access-log-destination** *acceptor*) *new-value*)
acceptor-document-root *acceptor => (or pathname null)*
(setf (**acceptor-document-root** *acceptor*) *new-value*)
acceptor-error-template-directory *acceptor => (or pathname null)*
(setf (**acceptor-error-template-directory** *acceptor*) *new-value*)
acceptor-input-chunking-p *acceptor => input-chunking-p*
(setf (**acceptor-input-chunking-p** *acceptor*) *new-value*)
acceptor-message-log-destination *acceptor => (or pathname null)*
(setf (**acceptor-message-log-destination** *acceptor*) *new-value*)
acceptor-name *acceptor => name*
(setf (**acceptor-name** *acceptor*) *new-value*)
acceptor-output-chunking-p *acceptor => output-chunking-p*
(setf (**acceptor-output-chunking-p** *acceptor*) *new-value*)
acceptor-persistent-connections-p *acceptor => persistent-connections-p*
(setf (**acceptor-persistent-connections-p** *acceptor*) *new-value*)
acceptor-reply-class *acceptor => reply-class*
(setf (**acceptor-reply-class** *acceptor*) *new-value*)
acceptor-request-class *acceptor => request-class*
(setf (**acceptor-request-class** *acceptor*) *new-value*)

These are accessors for various slots of ACCEPTOR objects. See the docstrings of these slots for more information and note that there are corresponding initargs for all of them.

[Generic function]
acceptor-ssl-p *acceptor => generalized-boolean*

Returns a true value if *acceptor* uses SSL connections. The default is to unconditionally return NIL and subclasses of ACCEPTOR must specialize this method to signal that they're using secure connections - see the SSL-ACCEPTOR class.

[Special variable]
default-connection-timeout

The default connection timeout used when an acceptor is reading from and writing to a socket stream. Note that some Lisps allow you to set different timeouts for reading and writing and you can specify both values via initargs when you create an acceptor.

[Generic function]
acceptor-remove-session *acceptor session => |*

This function is called whenever a session in ACCEPTOR is being destroyed because of a session timout or an explicit REMOVE-SESSION call.

Customizing acceptor behaviour

If you want to modify what acceptors do, you should subclass ACCEPTOR (or SSL-ACCEPTOR) and specialize the generic functions that constitute their behaviour (see example below). The life of an acceptor looks like this: It is started with the function START which immediately calls START-LISTENING and then applies the function EXECUTE-ACCEPTOR to its taskmaster. This function will eventually call ACCEPT-CONNECTIONS which is responsible for setting things up to wait for clients to connect. For each incoming connection which comes in, HANDLE-INCOMING-CONNECTION is applied to the taskmaster which will either call PROCESS-CONNECTION directly, or will create a thread to call it. PROCESS-CONNECTION calls INITIALIZE-CONNECTION-STREAM before it does anything else, then it selects and calls a function which handles the request, and finally it sends the reply to the client before it calls RESET-CONNECTION-STREAM. If the connection is persistent, this procedure is repeated (except for the intialization step) in a loop until the connection is closed. The acceptor is stopped with STOP.

If you just want to use the standard acceptors that come with Hunchentoot, you don't need to know anything about the functions listed in this section.

[Generic function]

start-listening *acceptor* => |

Sets up a listen socket for the given acceptor and enables it to listen to incoming connections. This function is called from the thread that starts the acceptor initially and may return errors resulting from the listening operation (like 'address in use' or similar).

[Generic function]
accept-connections *acceptor* => *nil*

In a loop, accepts a connection and hands it over to the acceptor's taskmaster for processing using HANDLE-INCOMING-CONNECTION. On LispWorks, this function returns immediately, on other Lisps it returns only once the acceptor has been stopped.

[Generic function]
process-connection *acceptor socket* => *nil*

This function is called by the taskmaster when a new client connection has been established. Its arguments are the ACCEPTOR object and a LispWorks socket handle or a usocket socket stream object in *socket*. It reads the request headers, sets up the request and reply objects, and hands over to PROCESS-REQUEST which calls HANDLE-REQUEST to select and call a handler for the request and sends its reply to the client. This is done in a loop until the stream has to be closed or until a connection timeout occurs. It is probably not a good idea to re-implement this method until you really, really know what you're doing.

Handlers may call to the DETACH-SOCKET generic function to indicate that no further requests should be handled on the connection by Hunchentoot, and that responsibility for the socket is assumed by third-party software. This can be used by specialized handlers that wish to hand over connection polling or processing to functions outside of Hunchentoot, i.e. for connection multiplexing or implementing specialized client protocols. Hunchentoot will finish processing the request and the PROCESS-CONNECTION function will return without closing the connection. At that point, the acceptor may interact with the socket in whatever fashion required.

[Generic function]
detach-socket *acceptor* => *stream*

Indicate to Hunchentoot that it should stop serving requests on the current request's socket. Hunchentoot will finish processing the current request and then return from PROCESS-CONNECTION without closing the connection to the client. DETACH-SOCKET can only be called from within a request handler function.

[Generic function]
initialize-connection-stream *acceptor stream* => *stream*

Can be used to modify the stream which is used to communicate between client and server before the request is read. The default method of ACCEPTOR does nothing, but see for example the method defined for SSL-ACCEPTOR. All methods of this generic function *must* return the stream to use.

[Generic function]
reset-connection-stream *acceptor stream* => *stream*

Resets the stream which is used to communicate between client and server after one request has been served so that it can be used to process the next request. This generic function is called after a request has been processed and *must* return the stream.

[Generic function]
acceptor-log-access *acceptor &key return-code*

Function to call to log access to the acceptor. The *return-code* keyword argument contains additional information about the request to log. In addition, it can use the standard request and reply accessor functions that are available to handler functions to find out more information about the request.

[Generic function]
acceptor-log-message *acceptor log-level format-string &rest format-arguments*

Function to call to log messages by the *acceptor*. It must accept a severity level for the message, which will be one of :ERROR, :INFO, or :WARNING, a format string and an arbitary number of formatting arguments.

[Generic function]
acceptor-status-message *acceptor http-return-code &key &allow-other-keys*

This function is called when a request's handler has been called but failed to provide content to send back to the client. It converts the *HTTP-STATUS-CODE* to some request contents, typically a human readable description of the status code to be displayed to the user. If an ERROR-TEMPLATE-DIRECTORY is set in the current acceptor and the directory contains a file corresponding to HTTP-STATUS-CODE named <code>.html, that file is sent to the client after variable substitution. Variables are referenced by ${<variable-name>}. Additional keyword arguments may be provided which are made available to the templating logic as substitution variables. These variables can be interpolated into error message templates in, which contains the current URL relative to the server and without GET parameters. In addition to the variables corresponding to keyword arguments, the script-name, lisp-implementation-type, lisp-implementation-version and hunchentoot-version variables are available.

An example of how to subclass ACCEPTOR

This example shows how to subclass ACCEPTOR in order to provide Hunchentoot with basic virtual host support. It assumes Hunchentoot is sitting behind an Internet-facing reverse-proxy web server that maps the host (or domain) part of incoming HTTP requests to unique localhost ports.

```
(asdf:load-system "hunchentoot")
(asdf:load-system "drakma")

;;; Subclass ACCEPTOR
(defclass vhost (hunchentoot:acceptor)
  ;; slots
  ((dispatch-table
    :initform '()
    :accessor dispatch-table
    :documentation "List of dispatch functions"))
  ;; options
  (:default-initargs                    ; default-initargs must be used
   :address "127.0.0.1"))               ; because ACCEPTOR uses it

;;; Specialise ACCEPTOR-DISPATCH-REQUEST for VHOSTs
(defmethod hunchentoot:acceptor-dispatch-request ((vhost vhost) request)
  ;; try REQUEST on each dispatcher in turn
  (mapc (lambda (dispatcher)
   (let ((handler (funcall dispatcher request)))
     (when handler              ; Handler found. FUNCALL it and return result
       (return-from hunchentoot:acceptor-dispatch-request (funcall handler)))))
 (dispatch-table vhost))
  (call-next-method))

;;; ===========================================================================
;;; Now all we need to do is test it

;;; Instantiate VHOSTs
(defvar vhost1 (make-instance 'vhost :port 50001))
(defvar vhost2 (make-instance 'vhost :port 50002))

;;; Populate each dispatch table
(push
 (hunchentoot:create-prefix-dispatcher "/foo" 'foo1)
 (dispatch-table vhost1))
(push
 (hunchentoot:create-prefix-dispatcher "/foo" 'foo2)
 (dispatch-table vhost2))

;;; Define handlers
(defun foo1 () "Hello")
(defun foo2 () "Goodbye")

;;; Start VHOSTs
(hunchentoot:start vhost1)
(hunchentoot:start vhost2)

;;; Make some requests
(drakma:http-request "http://127.0.0.1:50001/foo")
;;; =|
;;; 127.0.0.1 - [2012-06-08 14:30:39] "GET /foo HTTP/1.1" 200 5 "-" "Drakma/1.2.6 (SBCL 1.0.56; Linux; 2.6.32-5-686; http://weitz.de/drakma/)"
;;; =>
;;; "Hello"
;;; 200
;;; ((:CONTENT-LENGTH . "5") (:DATE . "Fri, 08 Jun 2012 14:30:39 GMT")
;;;  (:SERVER . "Hunchentoot 1.2.3") (:CONNECTION . "Close")
;;;  (:CONTENT-TYPE . "text/html; charset=utf-8"))
;;; #<PURI:URI http://127.0.0.1:50001/foo>
;;; #<FLEXI-STREAMS:FLEXI-IO-STREAM {CA90059}>
;;; T
;;; "OK"
(drakma:http-request "http://127.0.0.1:50002/foo")
;;; =|
;;; 127.0.0.1 - [2012-06-08 14:30:47] "GET /foo HTTP/1.1" 200 7 "-" "Drakma/1.2.6 (SBCL 1.0.56; Linux; 2.6.32-5-686; http://weitz.de/drakma/)"
;;; =>
;;; "Goodbye"
;;; 200
;;; ((:CONTENT-LENGTH . "7") (:DATE . "Fri, 08 Jun 2012 14:30:47 GMT")
;;;  (:SERVER . "Hunchentoot 1.2.3") (:CONNECTION . "Close")
;;;  (:CONTENT-TYPE . "text/html; charset=utf-8"))
;;; #<PURI:URI http://127.0.0.1:50002/foo>
;;; #<FLEXI-STREAMS:FLEXI-IO-STREAM {CAE8059}>
;;; T
;;; "OK"
```

How to make each VHOST write to separate access log streams (or files) is left as an exercise to the reader.

Taskmasters

As a "normal" Hunchentoot user, you can completely ignore taskmasters and skip this section. But if you're still reading, here are the dirty details: Each acceptor has a taskmaster associated with it at creation time. It is the taskmaster's job to distribute the

work of accepting and handling incoming connections. The acceptor calls the taskmaster if appropriate and the taskmaster calls back into the acceptor. This is done using the generic functions described in this and the previous section. Hunchentoot comes with two standard taskmaster implementations - one (which is the default used on multi-threaded Lisps) which starts a new thread for each incoming connection and one which handles all requests sequentially. It should for example be relatively straightforward to create a taskmaster which allocates threads from a fixed pool instead of creating a new one for each connection.

You can control the resources consumed by a threaded taskmaster via two initargs. :max-thread-count lets you set the maximum number of request threads that can be processes simultaneously. If this is nil, the is no thread limit imposed. :max-accept-count lets you set the maximum number of requests that can be outstanding (i.e. being processed or queued for processing). If :max-thread-count is supplied and :max-accept-count is NIL, then a +HTTP-SERVICE-UNAVAILABLE+ error will be generated if there are more than the max-thread-count threads processing requests. If both :max-thread-count and :max-accept-count are supplied, then max-thread-count must be less than max-accept-count; if more than max-thread-count requests are being processed, then requests up to max-accept-count will be queued until a thread becomes available. If more than max-accept-count requests are outstanding, then a +HTTP-SERVICE-UNAVAILABLE+ error will be generated. In a load-balanced environment with multiple Hunchentoot servers, it's reasonable to provide :max-thread-count but leave :max-accept-count null. This will immediately result in +HTTP-SERVICE-UNAVAILABLE+ when one server is out of resources, so the load balancer can try to find another server. In an environment with a single Hunchentoot server, it's reasonable to provide both :max-thread-count and a somewhat larger value for :max-accept-count. This will cause a server that's almost out of resources to wait a bit; if the server is completely out of resources, then the reply will be +HTTP-SERVICE-UNAVAILABLE+. The default for these values is 100 and 120, respectively.

If you want to implement your own taskmasters, you should subclass TASKMASTER or one of its subclasses, SINGLE-THREADED-TASKMASTER or ONE-THREAD-PER-CONNECTION-TASKMASTER, and specialize the generic functions in this section.

[Standard class]
taskmaster

> An instance of this class is responsible for distributing the work of handling requests for its acceptor. This is an "abstract" class in the sense that usually only instances of subclasses of TASKMASTER will be used.

[Standard class]
one-thread-per-connection-taskmaster

> A taskmaster that starts one thread for listening to incoming requests and one thread for each incoming connection.
>
> This is the default taskmaster implementation for multi-threaded Lisp implementations.

[Standard class]
single-threaded-taskmaster

> A taskmaster that runs synchronously in the thread where the START function was invoked (or in the case of LispWorks in the thread started by COMM:START-UP-SERVER). This is the simplest possible taskmaster implementation in that its methods do nothing but calling their acceptor "sister" methods - EXECUTE-ACCEPTOR calls ACCEPT-CONNECTIONS, HANDLE-INCOMING-CONNECTION calls PROCESS-CONNECTION.

[Standard class]
multi-threaded-taskmaster

> This is an abstract class for taskmasters that use multiple threads; it is not a concrete class and you should not instantiate it with MAKE-INSTANCE. Instead, you should instantiate its subclass ONE-THREAD-PER-CONNECTION-TASKMASTER described above. MULTI-THREADED-TASKMASTER is intended to be inherited from by extensions to Hunchentoot, such as quux-hunchentoot's THREAD-POOLING-TASKMASTER, though at the moment, doing so only inherits one slot and one method, on EXECUTE-ACCEPTOR, to have it start a new thread for the acceptor, then saved in said slot.

[Generic function]
execute-acceptor *taskmaster => result*

> This is a callback called by the acceptor once it has performed all initial processing to start listening for incoming connections (see START-LISTENING). It usually calls the ACCEPT-CONNECTIONS method of the acceptor, but depending on the taskmaster instance the method might be called from a new thread.

[Generic function]
handle-incoming-connection *taskmaster socket => result*

> This function is called by the acceptor to start processing of requests on a new incoming connection. *socket* is the usocket instance that represents the new connection (or a socket handle on LispWorks). The taskmaster starts processing requests on the incoming connection by calling the PROCESS-CONNECTION method of the acceptor instance. The *socket* argument is passed to PROCESS-CONNECTION as an argument. If the taskmaster is a multi-threaded taskmaster, HANDLE-INCOMING-THREAD will call CREATE-REQUEST-HANDLER-THREAD, which will call PROCESS-CONNECTION in a new thread. HANDLE-INCOMING-THREAD might issue a +HTTP-SERVICE-UNAVAILABLE+ error if there are too many request threads or it might block waiting for a request thread to finish.

[Generic function]
start-thread *taskmaster thunk &key => thread*

> This function is a callback that starts a new thread that will call the given *thunk* in the context of the proper *taskmaster*, with appropriate context-dependent keyword arguments. ONE-THREAD-PER-CONNECTION-TASKMASTER uses it in

EXECUTE-ACCEPTOR and CREATE-REQUEST-HANDLER-THREAD, but specialized taskmasters may define more functions that use it. By default, it just creates a thread calling the thunk with a specified *name* keyword argument. Specialized taskmasters may wrap special bindings and condition handlers around the thunk call, register the thread in a management table, etc.

[Generic function]
create-request-handler-thread *taskmaster socket => thread*

This function is called by HANDLE-INCOMING-THREAD to create a new thread which calls PROCESS-CONNECTION. If you specialize this function, you must be careful to have the thread call DECREMENT-TASKMASTER-REQUEST-COUNT before it exits. A typical method will look like this:

```
(defmethod create-request-handler-thread ((taskmaster monitor-taskmaster) socket)
  (bt:make-thread
   (lambda ()
     (with-monitor-error-handlers
       (unwind-protect
           (with-monitor-variable-bindings
               (process-connection (taskmaster-acceptor taskmaster) socket))
         (decrement-taskmaster-request-count taskmaster))))))
```

[Generic function]
shutdown *taskmaster => taskmaster*

Shuts down the taskmaster, i.e. frees all resources that were set up by it. For example, a multi-threaded taskmaster might terminate all threads that are currently associated with it. This function is called by the acceptor's STOP method.

[Generic accessor]
taskmaster-acceptor *taskmaster => acceptor*
(setf (**taskmaster-acceptor** *taskmaster*) *new-value*)

This is an accessor for the slot of a TASKMASTER object that links back to the acceptor it is associated with.

Request dispatch and handling

The main job of HANDLE-REQUEST is to select and call a function which handles the request, i.e. which looks at the data the client has sent and prepares an appropriate reply to send back. This is by default implemented as follows:

The ACCEPTOR class defines a ACCEPTOR-DISPATCH-REQUEST generic function which is used to actually dispatch the request. This function is called by the default method of HANDLE-REQUEST. Each ACCEPTOR-DISPATCH-REQUEST method looks at the request object and depending on its contents decides to either handle the request or call the next method.

In order to dispatch a request, Hunchentoot calls the ACCEPTOR-DISPATCH-REQUEST generic functions. The method for ACCEPTOR tries to serve a static file relative to it's ACCEPTOR-DOCUMENT-ROOT. Application specific acceptor subclasses will typically perform URL parsing and dispatching according to the policy that is required.

The default method of HANDLE-REQUEST sets up standard logging and error handling before it calls the acceptor's request dispatcher.

Request handlers do their work by modifying the reply object if necessary and by eventually returning the response body in the form of a string or a binary sequence. As an alternative, they can also call SEND-HEADERS and write directly to a stream.

Using the easy-handler framework

The EASY-ACCEPTOR class defines a method for ACCEPTOR-DISPATCH-REQUEST that walks through the list *DISPATCH-TABLE* which consists of *dispatch functions*. Each of these functions accepts the request object as its only argument and either returns a request handler to handle the request or NIL which means that the next dispatcher in the list will be tried. A *request handler* is a function of zero arguments which relies on the special variable *REQUEST* to access the request instance being serviced. If all dispatch functions return NIL, the next ACCEPTOR-DISPATCH-REQUEST will be called.

N.B. All functions and variables in this section are related to the easy request dispatch mechanism and are meaningless if you're using your own request dispatcher.

[Standard class]
easy-acceptor

This class defines no additional slots with respect to ACCEPTOR. It only serves as an additional type for dispatching calls to ACCEPTOR-DISPATCH-REQUEST. In order to use the easy handler framework, acceptors of this class or one of its subclasses must be used.

[Standard class]
easy-ssl-acceptor

This class mixes the SSL-ACCEPTOR and the EASY-ACCEPTOR classes. It is used when both ssl and the easy handler framework are required.

[Special variable]

dispatch-table

A global list of dispatch functions. The initial value is a list consisting of the symbol DISPATCH-EASY-HANDLERS.

[Function]
create-prefix-dispatcher *prefix handler => dispatch-fn*

A convenience function which will return a dispatcher that returns *handler* whenever the path part of the request URI starts with the string *prefix*.

[Function]
create-regex-dispatcher *regex handler => dispatch-fn*

A convenience function which will return a dispatcher that returns *handler* whenever the path part of the request URI matches the CL-PPCRE regular expression *regex* (which can be a string, an s-expression, or a scanner).

[Function]
create-folder-dispatcher-and-handler *uri-prefix base-path &optional content-type => dispatch-fn*

Creates and returns a dispatch function which will dispatch to a handler function which emits the file relative to *base-path* that is denoted by the URI of the request relative to *uri-prefix*. *uri-prefix* must be a string ending with a slash, *base-path* must be a pathname designator for an existing directory. Uses HANDLE-STATIC-FILE internally.

If *content-type* is *not* NIL, it will be used as a the content type for all files in the folder. Otherwise (which is the default) the content type of each file will be determined as usual.

[Function]
create-static-file-dispatcher-and-handler *uri path &optional content-type => result*

Creates and returns a request dispatch function which will dispatch to a handler function which emits the file denoted by the pathname designator PATH with content type CONTENT-TYPE if the SCRIPT-NAME of the request matches the string URI. If CONTENT-TYPE is NIL, tries to determine the content type via the file's suffix.

[Macro]
define-easy-handler *description lambda-list [[declaration* | documentation]] form**

Defines a handler as if by DEFUN and optionally registers it with a URI so that it will be found by DISPATCH-EASY-HANDLERS.

description is either a symbol *name* or a list matching the destructuring lambda list

```
(name &key uri acceptor-names default-parameter-type default-request-type).
```

lambda-list is a list the elements of which are either a symbol *var* or a list matching the destructuring lambda list

```
(var &key real-name parameter-type init-form request-type).
```

The resulting handler will be a Lisp function with the name *name* and keyword parameters named by the *var* symbols. Each *var* will be bound to the value of the GET or POST parameter called *real-name* (a string) before the body of the function is executed. If *real-name* is not provided, it will be computed by downcasing the symbol name of *var*.

If *uri* (which is evaluated) is provided, then it must be a string or a function designator for a unary function. In this case, the handler will be returned by DISPATCH-EASY-HANDLERS, if *uri* is a string and the script name of the current request is *uri*, or if *uri* designates a function and applying this function to the current REQUEST object returns a true value.

acceptor-names (which is evaluated) can be a list of symbols which means that the handler will only be returned by DISPATCH-EASY-HANDLERS in acceptors which have one of these names (see ACCEPTOR-NAME). *acceptor-names* can also be the symbol T which means that the handler will be returned by DISPATCH-EASY-HANDLERS in *every* acceptor.

Whether the GET or POST parameter (or both) will be taken into consideration, depends on *request-type* which can be :GET, :POST, :BOTH, or NIL. In the last case, the value of *default-request-type* (the default of which is :BOTH) will be used.

The value of *var* will usually be a string (unless it resulted from a file upload in which case it won't be converted at all), but if *parameter-type* (which is evaluated) is provided, the string will be converted to another Lisp type by the following rules:

If the corresponding GET or POST parameter wasn't provided by the client, *var*'s value will be NIL. If *parameter-type* is 'STRING, *var*'s value remains as is. If *parameter-type* is 'INTEGER and the parameter string consists solely of decimal digits, *var*'s value will be the corresponding integer, otherwise NIL. If *parameter-type* is 'KEYWORD, *var*'s value will be the keyword obtained by interning the upcased parameter string into the keyword package. If *parameter-type* is 'CHARACTER and the parameter string is of length one, *var*'s value will be the single character of this string, otherwise NIL. If *parameter-type* is 'BOOLEAN, *var*'s value will always be T (unless it is NIL by the first rule above, of course). If *parameter-type* is any other atom, it is supposed to be a function designator for a unary function which will be called to convert the string to something else.

Those were the rules for *simple* parameter types, but *parameter-type* can also be a list starting with one of the symbols LIST, ARRAY, or HASH-TABLE. The second value of the list must always be a simple parameter type as in the last paragraph - we'll call it the *inner type* below.

In the case of 'LIST, all GET/POST parameters called *real-name* will be collected, converted to the inner type as by the rules above, and assembled into a list which will be the value of *var*.

In the case of 'ARRAY, all GET/POST parameters which have a name like the result of

```
(format nil "~A[~A]" real-name n)
```

where *n* is a non-negative integer, will be assembled into an array where the *n*th element will be set accordingly, after conversion to the inner type. The array, which will become the value of *var*, will be big enough to hold all matching parameters, but not bigger. Array elements not set as described above will be NIL. Note that VAR will always be bound to an array, which may be empty, so it will never be NIL, even if no appropriate GET/POST parameters are found.

The full form of a 'HASH-TABLE parameter type is

```
(hash-table inner-type key-type test-function)
```

but *key-type* and *test-function* can be left out in which case they default to 'STRING and 'EQUAL, respectively. For this parameter type, all GET/POST parameters which have a name like the result of

```
(format nil "~A{~A}" real-name key)
```

(where *key* is a string that doesn't contain curly brackets) will become the values (after conversion to *inner-type*) of a hash table with test function *test-function* where *key* (after conversion to *key-type*) will be the corresponding key. Note that *var* will always be bound to a hash table, which may be empty, so it will never be NIL, even if no appropriate GET/POST parameters are found.

To make matters even more complicated, the three compound parameter types also have an abbreviated form - just one of the symbols LIST, ARRAY, or HASH-TABLE. In this case, the inner type will default to 'STRING.

If *parameter-type* is not provided or NIL, *default-parameter-type* (the default of which is 'STRING) will be used instead.

If the result of the computations above would be that *var* would be bound to NIL, then *init-form* (if provided) will be evaluated instead, and *var* will be bound to the result of this evaluation.

Handlers built with this macro are constructed in such a way that the resulting Lisp function is useful even outside of Hunchentoot. Specifically, all the parameter computations above will only happen if *REQUEST* is bound, i.e. if we're within a Hunchentoot request. Otherwise, *var* will always be bound to the result of evaluating *init-form* unless a corresponding keyword argument is provided.

The example code that comes with Hunchentoot contains an example which demonstrates some of the features of DEFINE-EASY-HANDLER.

[Function]
dispatch-easy-handlers *request => result*

This is a dispatcher which returns the appropriate handler defined with DEFINE-EASY-HANDLER, if there is one.

Request objects

For each incoming request, the acceptor (in PROCESS-CONNECTION) creates a REQUEST object and makes it available to handlers via the special variable *REQUEST*. This object contains all relevant information about the request and this section collects the functions which can be used to query such an object. In all function where *request* is an optional or keyword parameter, the default is *REQUEST*.

If you need more fine-grained control over the behaviour of request objects, you can subclass REQUEST and initialize the REQUEST-CLASS slot of the ACCEPTOR class accordingly. The acceptor will generate request objects of the class named by this slot.

[Standard class]
request

Objects of this class hold all the information about an incoming request. They are created automatically by acceptors and can be accessed by the corresponding handler. You should not mess with the slots of these objects directly, but you can subclass REQUEST in order to implement your own behaviour. See the REQUEST-CLASS slot of the ACCEPTOR class.

[Special variable]
request

The current REQUEST object while in the context of a request.

[Function]
real-remote-addr *&optional request => string{, list}*

Returns the 'X-Forwarded-For' incoming http header as the second value in the form of a list of IP addresses and the first element of this list as the first value if this header exists. Otherwise returns the value of REMOTE-ADDR as the only

value.

[Function]
parameter *name* *&optional* *request* => *string*

Returns the GET or the POST parameter with name *name* (a string) - or NIL if there is none. If both a GET and a POST parameter with the same name exist the GET parameter is returned. Search is case-sensitive. See also GET-PARAMETER and POST-PARAMETER.

[Function]
get-parameter *name* *&optional* *request* => *string*

Returns the value of the GET parameter (as provided via the request URI) named by the string *name* as a string (or NIL if there ain't no GET parameter with this name). Note that only the first value will be returned if the client provided more than one GET parameter with the name *name*. See also GET-PARAMETERS*.

[Function]
post-parameter *name* *&optional* *request* => *string*

Returns the value of the POST parameter (as provided in the request's body) named by the string *name*. Note that only the first value will be returned if the client provided more than one POST parameter with the name *name*. This value will usually be a string (or NIL if there ain't no POST parameter with this name). If, however, the browser sent a file through a multipart/form-data form, the value of this function is a three-element list

```
(path file-name content-type)
```

where *path* is a pathname denoting the place were the uploaded file was stored, *file-name* (a string) is the file name sent by the browser, and *content-type* (also a string) is the content type sent by the browser. The file denoted by *path* will be deleted after the request has been handled - you have to move or copy it somewhere else if you want to keep it.

POST parameters will only be computed if the content type of the request body was multipart/form-data or application/x-www-form-urlencoded. Although this function is called POST-PARAMETER, you can instruct Hunchentoot to compute these parameters for other request methods by setting *METHODS-FOR-POST-PARAMETERS*.

See also POST-PARAMETERS and *TMP-DIRECTORY*.

[Function]
get-parameters* *&optional* *request* => *alist*

Returns an alist of all GET parameters (as provided via the request URI). The car of each element of this list is the parameter's name while the cdr is its value (as a string). The elements of this list are in the same order as they were within the request URI. See also GET-PARAMETER.

[Function]
post-parameters* *&optional* *request* => *alist*

Returns an alist of all POST parameters (as provided via the request's body). The car of each element of this list is the parameter's name while the cdr is its value. The elements of this list are in the same order as they were within the request's body.

See also POST-PARAMETER.

[Special variable]
methods-for-post-parameters

A list of the request method types (as keywords) for which Hunchentoot will try to compute *post-parameters*.

[Function]
cookie-in *name* *&optional* *request* => *string*

Returns the cookie with the name *name* (a string) as sent by the browser - or NIL if there is none.

[Function]
cookies-in* *&optional* *request* => *alist*

Returns an alist of all cookies associated with the REQUEST object *request*.

[Function]
host *&optional* *request* => *host*

Returns the 'Host' incoming http header value.

[Function]
query-string* *&optional* *request* => *string*

Returns the query string of the REQUEST object *request*. That's the part behind the question mark (i.e. the GET parameters).

[Function]
referer *&optional request => result*

Returns the 'Referer' (sic!) http header.

[Function]
request-method* *&optional request => keyword*

Returns the request method as a Lisp keyword.

[Function]
request-uri* *&optional request => uri*

Returns the request URI.

[Function]
server-protocol* *&optional request => keyword*

Returns the request protocol as a Lisp keyword.

[Function]
user-agent *&optional request => result*

Returns the 'User-Agent' http header.

[Function]
header-in* *name &optional request => header*

Returns the incoming header with name *name*. *name* can be a keyword (recommended) or a string.

[Function]
headers-in* *&optional request => alist*

Returns an alist of the incoming headers associated with the REQUEST object *request*.

[Function]
remote-addr* *&optional request => address*

Returns the address the current request originated from.

[Function]
remote-port* *&optional request => port*

Returns the port the current request originated from.

[Function]
local-addr* *&optional request => address*

The IP address of the local system that the client connected to.

[Function]
local-port* *&optional request => port*

The TCP port number of the local system that the client connected to.

[Function]
script-name* *&optional request => script-name*

Returns the file name of the REQUEST object *request*. That's the requested URI without the query string (i.e the GET parameters).

[Accessor]
aux-request-value *symbol &optional request => value, present-p*
(setf (**aux-request-value** *symbol &optional request*) *new-value*)

This accessor can be used to associate arbitrary data with the the symbol *symbol* in the REQUEST object *request*. *present-p* is true if such data was found, otherwise NIL.

[Function]
delete-aux-request-value *symbol &optional request => |*

Removes the value associated with *symbol* from the REQUEST object *request*.

[Function]
authorization *&optional request => result*

Returns as two values the user and password (if any) as encoded in the 'AUTHORIZATION' header. Returns NIL if there is no such header.

[Special variable]
hunchentoot-default-external-format

The external format used to compute the REQUEST object.

[Special variable]
file-upload-hook

If this is not NIL, it should be a unary function which will be called with a pathname for each file which is uploaded to Hunchentoot. The pathname denotes the temporary file to which the uploaded file is written. The hook is called directly before the file is created. At this point, *REQUEST* is already bound to the current REQUEST object, but obviously you can't access the post parameters yet.

[Function]
raw-post-data *&key request external-format force-text force-binary want-stream => raw-body-or-stream*

Returns the content sent by the client in the request body if there was any (unless the content type was multipart/form-data in which case NIL is returned). By default, the result is a string if the type of the Content-Type media type is "text", and a vector of octets otherwise. In the case of a string, the external format to be used to decode the content will be determined from the charset parameter sent by the client (or otherwise *HUNCHENTOOT-DEFAULT-EXTERNAL-FORMAT* will be used).

You can also provide an external format explicitly (through *external-format*) in which case the result will unconditionally be a string. Likewise, you can provide a true value for *force-text* which will force Hunchentoot to act as if the type of the media type had been "text" (with *external-format* taking precedence if provided). Or you can provide a true value for *force-binary* which means that you want a vector of octets at any rate. (If both *force-text* and *force-binary* are true, an error will be signaled.)

If, however, you provide a true value for *want-stream*, the other parameters are ignored and you'll get the content (flexi) stream to read from it yourself. It is then your responsibility to read the correct amount of data, because otherwise you won't be able to return a response to the client. The stream will have its octet position set to 0. If the client provided a Content-Length header, the stream will also have a corresponding bound, so no matter whether the client used chunked encoding or not, you can always read until EOF.

If the content type of the request was multipart/form-data or application/x-www-form-urlencoded, the content has been read by Hunchentoot already and you can't read from the stream anymore.

You can call RAW-POST-DATA more than once per request, but you can't mix calls which have different values for *want-stream*.

Note that this function is slightly misnamed because a client can send content even if the request method is not POST.

[Function]
recompute-request-parameters *&key request external-format => |*

Recomputes the GET and POST parameters for the REQUEST object *request*. This only makes sense if you're switching external formats during the request.

[Generic function]
process-request *request => nil*

This function is called by PROCESS-CONNECTION after the incoming headers have been read. It calls HANDLE-REQUEST (and is more or less just a thin wrapper around it) to select and call a handler and send the output of this handler to the client. Note that PROCESS-CONNECTION is called once per connection and loops in case of a persistent connection while PROCESS-REQUEST is called anew for each request.

The return value of this function is ignored.

Like PROCESS-CONNECTION, this is another function the behaviour of which you should only modify if you really, really know what you're doing.

[Generic function]
handle-request *acceptor request => content*

This function is called by PROCESS-REQUEST once the request has been read and a REQUEST object has been created. Its job is to actually handle the request, i.e. to return something to the client.

The default method calls the acceptor's request dispatcher, but you can of course implement a different behaviour. The default method also sets up standard error handling for the handler.

Might be a good place to bind or rebind special variables which can then be accessed by your handlers.

[Generic function]
acceptor-dispatch-request *acceptor request => content*

This function is called to actually dispatch the request once the standard logging and error handling has been set up. ACCEPTOR subclasses implement methods for this function in order to perform their own request routing. If a method does not want to handle the request, it is supposed to invoke CALL-NEXT-METHOD so that the next ACCEPTOR in

the inheritance chain gets a chance to handle the request.

[Generic readers]
cookies-in *request => cookies*
get-parameters *request => get-parameters*
header-in *name request => result*
headers-in *request => headers*
post-parameters *request => post-parameters*
query-string *request => query-string*
remote-addr *request => address*
remote-port *request => port*
local-addr *request => address*
local-port *request => port*
request-acceptor *request => acceptor*
request-method *request => method*
request-uri *request => uri*
server-protocol *request => protocol*
script-name *request => result*

These are various generic readers which are used to read information about a REQUEST object. If you are writing a handler, you should *not* use these readers but instead utilize the corresponding functions with an asterisk at the end of their name, also listed in this section. These generic readers are only exported for users who want to create their own subclasses of REQUEST.

Reply objects

For each incoming request, the acceptor (in PROCESS-CONNECTION) creates a REPLY object and makes it available to handlers via the special variable *REPLY*. This object contains all relevant information (except for the content body) about the reply that will be sent to the client and this section collects the functions which can be used to query and modify such an object. In all function where *reply* is an optional or keyword parameter, the default is *REPLY*.

If you need more fine-grained control over the behaviour of reply objects, you can subclass REPLY and initialize the REPLY-CLASS slot of the ACCEPTOR class accordingly. The acceptor will generate reply objects of the class named by this slot.

[Standard class]
reply

Objects of this class hold all the information about an outgoing reply. They are created automatically by Hunchentoot and can be accessed and modified by the corresponding handler.

You should not mess with the slots of these objects directly, but you can subclass REPLY in order to implement your own behaviour. See the :reply-class initarg of the ACCEPTOR class.

[Special variable]
reply

The current REPLY object in the context of a request.

[Accessor]
header-out *name &optional reply => string*
(setf (**header-out** *name &optional reply*) *new-value*)

HEADER-OUT returns the outgoing http header named by the keyword *name* if there is one, otherwise NIL. SETF of HEADER-OUT changes the current value of the header named *name*. If no header named *name* exists, it is created. For backwards compatibility, *name* can also be a string in which case the association between a header and its name is case-insensitive.

Note that the header 'Set-Cookie' cannot be queried by HEADER-OUT and must not be set by SETF of HEADER-OUT. See also HEADERS-OUT*, CONTENT-TYPE*, CONTENT-LENGTH*, COOKIES-OUT*, and COOKIE-OUT.

[Function]
headers-out* *&optional reply => alist*

Returns an alist of the outgoing headers associated with the REPLY object *reply*. See also HEADER-OUT.

[Accessor]
content-length* *&optional reply => content-length*
(setf (**content-length*** *&optional reply*) *new-value*)

The outgoing 'Content-Length' http header of *reply*.

[Accessor]
content-type* *&optional reply => content-type*
(setf (**content-type*** *&optional reply*) *new-value*)

The outgoing 'Content-Type' http header of *reply*.

[Function]
cookie-out *name &optional reply => result*

Returns the current value of the outgoing cookie named *name*. Search is case-sensitive.

[Accessor]
cookies-out* *&optional reply => alist*
(setf (**cookies-out*** *&optional reply*) *new-value*)

Returns or sets an alist of the outgoing cookies associated with the REPLY object *reply*.

[Accessor]
return-code* *&optional reply => return-code*
(setf (**return-code*** *&optional reply*) *new-value*)

Gets or sets the http return code of *reply*. The return code of each REPLY object is initially set to +HTTP-OK+.

[Function]
send-headers *=> stream*

Sends the initial status line and all headers as determined by the REPLY object *REPLY*. Returns a binary stream to which the body of the reply can be written. Once this function has been called, further changes to *REPLY* don't have any effect. Also, automatic handling of errors (i.e. sending the corresponding status code to the browser, etc.) is turned off for this request and functions like REDIRECT or to ABORT-REQUEST-HANDLER won't have the desired effect once the headers are sent.

If your handlers return the full body as a string or as an array of octets, you should *not* call this function. If a handler calls SEND-HEADERS, its return value is ignored.

[Accessor]
reply-external-format* *&optional reply => external-format*
(setf (**reply-external-format*** *&optional reply*) *new-value*)

Gets or sets the external format of *reply* which is used for character output.

[Special variable]
default-content-type

The default content-type header which is returned to the client.

[Constants]
+http-continue+
+http-switching-protocols+
+http-ok+
+http-created+
+http-accepted+
+http-non-authoritative-information+
+http-no-content+
+http-reset-content+
+http-partial-content+
+http-multi-status+
+http-multiple-choices+
+http-moved-permanently+
+http-moved-temporarily+
+http-see-other+
+http-not-modified+
+http-use-proxy+
+http-temporary-redirect+
+http-bad-request+
+http-authorization-required+
+http-payment-required+
+http-forbidden+
+http-not-found+
+http-method-not-allowed+
+http-not-acceptable+
+http-proxy-authentication-required+
+http-request-time-out+
+http-conflict+
+http-gone+
+http-length-required+
+http-precondition-failed+
+http-request-entity-too-large+
+http-request-uri-too-large+
+http-unsupported-media-type+
+http-requested-range-not-satisfiable+
+http-expectation-failed+
+http-failed-dependency+

+http-internal-server-error+
+http-not-implemented+
+http-bad-gateway+
+http-service-unavailable+
+http-gateway-time-out+
+http-version-not-supported+

The values of these constants are 100, 101, 200, 201, 202, 203, 204, 205, 206, 207, 300, 301, 302, 303, 304, 305, 307, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 424, 500, 501, 502, 503, 504, and 505. See RETURN-CODE.

[Generic readers]
content-length *reply* => *content-length*
content-type *reply* => *content-type*
headers-out *reply* => *headers-out*

These are various generic readers which are used to read information about a REPLY object. If you are writing a handler, you should *not* use these readers but instead utilize the corresponding functions with an asterisk at the end of their name, also listed in this section. These generic readers are only exported for users who want to create their own subclasses of REPLY.

[Generic accessors]
cookies-out *reply* => *result*
(setf (**cookies-out** *reply*) *new-value*)
return-code *reply* => *result*
(setf (**return-code** *reply*) *new-value*)
reply-external-format *reply* => *result*
(setf (**reply-external-format** *reply*) *new-value*)

These are various generic accessors which are used to query and modify a REPLY objects. If you are writing a handler, you should *not* use these accessors but instead utilize the corresponding functions with an asterisk at the end of their name, also listed in this section. These generic accessors are only exported for users who want to create their own subclasses of REPLY.

Sessions

Hunchentoot supports *sessions*: Once a request handler has called START-SESSION, Hunchentoot uses either cookies or (if the client doesn't send the cookies back) rewrites URLs to keep track of this client, i.e. to provide a kind of 'state' for the stateless http protocol. The session associated with the client is a CLOS object which can be used to store arbitrary data between requests.

Hunchentoot makes some reasonable effort to prevent eavesdroppers from hijacking sessions (see below), but this should not be considered really secure. Don't store sensitive data in sessions and rely solely on the session mechanism as a safeguard against malicious users who want to get at this data!

For each request there's one SESSION object which is accessible to the handler via the special variable *SESSION*. This object holds all the information available about the session and can be accessed with the functions described in this chapter. Note that the internal structure of SESSION objects should be considered opaque and may change in future releases of Hunchentoot.

Sessions are automatically verified for validity and age when the REQUEST object is instantiated, i.e. if *SESSION* is not NIL then this session is valid (as far as Hunchentoot is concerned) and not too old. Old sessions are automatically removed.

Hunchentoot also provides a SESSION-REGENERATE-COOKIE-VALUE function that creates a new cookie value. This helps to prevent against session fixation attacks, and should be used when a user logs in according to the application.

[Standard class]
session

SESSION objects are automatically maintained by Hunchentoot. They should not be created explicitly with MAKE-INSTANCE but implicitly with START-SESSION and they should be treated as opaque objects.

You can ignore Hunchentoot's SESSION objects and implement your own sessions if you provide corresponding methods for SESSION-COOKIE-VALUE and SESSION-VERIFY.

[Function]
start-session => *session*

Returns the current SESSION object. If there is no current session, creates one and updates the corresponding data structures. In this case the function will also send a session cookie to the browser.

[Accessor]
session-value *symbol* *&optional* *session* => *value, present-p*
(setf (**session-value** *symbol* *&optional* *session*) *new-value*)

This accessor can be used to associate arbitrary data with the the symbol *symbol* in the SESSION object *session*. *present-p* is true if such data was found, otherwise NIL. The default value for *session* is *SESSION*.

If SETF of SESSION-VALUE is called with *session* being NIL then a session is automatically instantiated with START-

SESSION.

[Function]
delete-session-value *symbol* *&optional* *session* => |

Removes the value associated with *symbol* from *session* if there is one.

[Special variable]
session

The current session while in the context of a request, or NIL.

[Function]
remove-session *session* => |

Completely removes the SESSION object *session* from Hunchentoot's internal session database.

[Function]
reset-sessions *&optional* *acceptor* => |

Removes *all* stored sessions of *acceptor*. The default for *acceptor* is *ACCEPTOR*.

[Function]
regenerate-session-cookie-value *session* => *cookie*

Regenerates the session cookie value. This should be used when a user logs in according to the application to prevent against session fixation attacks. The cookie value being dependent on ID, USER-AGENT, REMOTE-ADDR, START, and *SESSION-SECRET*, the only value we can change is START to regenerate a new value. Since we're generating a new cookie, it makes sense to have the session being restarted, in time. That said, because of this fact, calling this function twice in the same second will regenerate twice the same value.

[Special variable]
rewrite-for-session-urls

Whether HTML pages should possibly be rewritten for cookie-less session-management.

[Special variable]
content-types-for-url-rewrite

The content types for which url-rewriting is OK. See *REWRITE-FOR-SESSION-URLS*.

[Special variable]
use-remote-addr-for-sessions

Whether the client's remote IP (as returned by REAL-REMOTE-ADDR) should be encoded into the session string. If this value is true, a session will cease to be accessible if the client's remote IP changes.

This might for example be an issue if the client uses a proxy server which doesn't send correct 'X-Forwarded-For' headers.

[Generic function]
session-remote-addr *session* => *remote-addr*

The remote IP address of the client when this session was started (as returned by REAL-REMOTE-ADDR).

[Special variable]
use-user-agent-for-sessions

Whether the 'User-Agent' header should be encoded into the session string. If this value is true, a session will cease to be accessible if the client sends a different 'User-Agent' header.

[Generic function]
session-user-agent *session* => *user-agent*

The incoming 'User-Agent' header that was sent when this session was created.

[Generic accessor]
session-max-time *session* => *max-time*
(setf (**session-max-time** *session*) *new-value*)

Gets or sets the time (in seconds) after which *session* expires if it's not used.

[Special variable]
session-max-time

The default time (in seconds) after which a session times out.

[Special variable]

session-gc-frequency

A session GC (see function SESSION-GC) will happen every *SESSION-GC-FREQUENCY* requests (counting only requests which create a new session) if this variable is not NIL. See SESSION-CREATED.

[Function]
session-gc => |

Removes sessions from the current session database which are too old - see SESSION-TOO-OLD-P.

[Function]
session-too-old-p *session => generalized-boolean*

Returns true if the SESSION object *session* has not been active in the last (session-max-time session) seconds.

[Generic function]
session-id *session => session-id*

The unique ID (an INTEGER) of the session.

[Generic function]
session-start *session => universal-time*

The time this session was started.

Customizing session behaviour

For everyday session usage, you will probably just use START-SESSION, SESSION-VALUE, and maybe DELETE-SESSION-VALUE and *SESSION*. However, there are two ways to customize the way Hunchentoot maintains sessions.

One way is to mostly leave the session mechanism intact but to tweak it a bit:

- The publicly visible part of a session is encoded using a secret which you can set yourself.
- And it is stored using a cookie (or GET parameter) name that you can override.
- Each session receives a new ID when it is created and you can implement a more robust way to do that.
- You can arrange to be called whenever a session is created to trigger some action. You might also do this to invent your own session garbage collection.
- By default, all sessions are stored in a global alist in memory. You can't change the alist part, but you can distribute your sessions over different "databases".
- By default, every operation which modifies sessions or one of the session databases is guarded by a global lock, but you can arrange to provide different locks for this.

The other way to customize Hunchentoot's sessions is to completely replace them. This is actually pretty easy: Create your own class to store state (which doesn't have to and probably shouldn't inherit from SESSION) and implement methods for SESSION-VERIFY and SESSION-COOKIE-VALUE - that's it. Hunchentoot will continue to use cookies and/or to rewrite URLs to keep track of session state and it will store "the current session" (whatever that is in your implementation) in *SESSION*. Everything else (like persisting sessions, GC, getting and setting values) you'll have to take care of yourself and the other session functions (like START-SESSION or SESSION-VALUE) won't work anymore. (Almost) total freedom, but a lot of responsibility as well... :)

[Special variable]
session-secret

A random ASCII string that's used to encode the public session data. This variable is initially unbound and will be set (using RESET-SESSION-SECRET) the first time a session is created, if necessary. You can prevent this from happening if you set the value yourself before starting acceptors.

[Function]
reset-session-secret => *secret*

Sets *SESSION-SECRET* to a new random value. All old sessions will cease to be valid.

[Generic function]
session-cookie-name *acceptor => name*

Returns the name (a string) of the cookie (or the GET parameter) which is used to store a session on the client side. The default is to use the string "hunchentoot-session", but you can specialize this function if you want another name.

[Generic function]
session-created *acceptor new-session => result*

This function is called whenever a new session has been created. There's a default method which might trigger a session GC based on the value of *SESSION-GC-FREQUENCY*.

The return value is ignored.

[Generic function]
next-session-id *acceptor => id*

Returns the next sequential session ID, an integer, which should be unique per session. The default method uses a simple global counter and isn't guarded by a lock. For a high-performance production environment you might consider using a more robust implementation.

[Generic accessor]
session-db *acceptor => database*
(setf (**session-db** *acceptor*) *new-value*)

Returns the current session database which is an alist where each car is a session's ID and the cdr is the corresponding SESSION object itself. The default is to use a global list for all acceptors.

[Generic function]
session-db-lock *acceptor &key whole-db-p => lock*

A function which returns a lock that will be used to prevent concurrent access to sessions. The first argument will be the acceptor that handles the current request, the second argument is true if the whole (current) session database is modified. If it is NIL, only one existing session in the database is modified.

This function can return NIL which means that sessions or session databases will be modified without a lock held (for example for single-threaded environments). The default is to always return a global lock (ignoring the *acceptor* argument) for Lisps that support threads and NIL otherwise.

[Generic function]
session-verify *request => session-or-nil*

Tries to get a session identifier from the cookies (or alternatively from the GET parameters) sent by the client (see SESSION-COOKIE-NAME and SESSION-COOKIE-VALUE). This identifier is then checked for validity against the REQUEST object *request*. On success the corresponding session object (if not too old) is returned (and updated). Otherwise NIL is returned.

A default method is provided and you only need to write your own one if you want to maintain your own sessions.

[Generic function]
session-cookie-value *session => string*

Returns a string which can be used to safely restore the session *session* if as session has already been established. This is used as the value stored in the session cookie or in the corresponding GET parameter and verified by SESSION-VERIFY.

A default method is provided and there's no reason to change it unless you want to use your own session objects.

Cookies

Outgoing cookies are stored in the request's REPLY object (see COOKIE-OUT and COOKIES-OUT*). They are CLOS objects defined like this:

```
(defclass cookie ()
  ((name :initarg :name
         :reader cookie-name
         :type string
         :documentation "The name of the cookie - a string.")
   (value :initarg :value
          :accessor cookie-value
          :initform ""
          :documentation "The value of the cookie. Will be URL-encoded when sent to the browser.")
   (expires :initarg :expires
            :initform nil
            :accessor cookie-expires
            :documentation "The time (a universal time) when the cookie expires (or NIL).")
   (max-age :initarg :max-age
            :initform nil
            :accessor cookie-max-age
            :documentation "The time delta (in seconds) after which the cookie expires (or NIL).")
   (path :initarg :path
         :initform nil
         :accessor cookie-path
         :documentation "The path this cookie is valid for (or NIL).")
   (domain :initarg :domain
           :initform nil
           :accessor cookie-domain
           :documentation "The domain this cookie is valid for (or NIL).")
   (secure :initarg :secure
           :initform nil
           :accessor cookie-secure
           :documentation "A generalized boolean denoting whether this is a secure cookie.")
   (http-only :initarg :http-only
              :initform nil
              :accessor cookie-http-only
              :documentation "A generalized boolean denoting whether this is a HttpOnly cookie.")))
```

The reader COOKIE-NAME and the accessors COOKIE-VALUE, COOKIE-EXPIRES, COOKIE-MAX-AGE, COOKIE-PATH, COOKIE-DOMAIN,

COOKIE-SECURE, and COOKIE-HTTP-ONLY are all exported from the HUNCHENTOOT package. For now, the class name itself is *not* exported.

[Function]
set-cookie *name &key value expires path domain secure http-only reply => cookie*

Creates a COOKIE object from the parameters provided to this function and adds it to the outgoing cookies of the REPLY object *reply*. If a cookie with the same name (case-sensitive) already exists, it is replaced. The default for *reply* is *REPLY*. The default for *value* is the empty string.

[Function]
set-cookie* *cookie &optional reply => cookie*

Adds the COOKIE object *cookie* to the outgoing cookies of the REPLY object *reply*. If a cookie with the same name (case-sensitive) already exists, it is replaced. The default for *reply* is *REPLY*.

Logging

Hunchentoot can log accesses and diagnostic messages to two separate destinations, which can be either files in the file system or streams. Logging can also be disabled by setting the ACCESS-LOG-DESTINATION and MESSAGE-LOG-DESTINATION slots in the ACCEPTOR to NIL. The two slots can be initialized by providing the :ACCESS-LOG-DESTINATION and :MESSAGE-LOG-DESTINATION initialization arguments when creating the acceptor or set by setting the slots through its ACCEPTOR-MESSAGE-LOG-DESTINATION and ACCEPTOR-ACCESS-LOG-DESTINATION accessors.

When the path for the message or accept log is set to a variable holding an output stream, hunchentoots writes corresponding log entries to that stream. By default, Hunchentoot logs to *STANDARD-ERROR*.

Access logging is done in a format similar to what the Apache web server can write so that logfile analysis using standard tools is possible. Errors during request processing are logged to a separate file.

The standard logging mechanism is deliberately simple and slow. The log files are opened for each log entry and closed again after writing, and access to them is protected by a global lock. Derived acceptor classes can implement methods for the ACCEPTOR-LOG-MESSAGE and ACCEPTOR-LOG-ACCESS generic functions in order to log differently (e.g. to a central logging server or in a different file format.

Errors happening within a handler which are not caught by the handler itself are handled by Hunchentoot by logging them to the established ACCEPTOR-MESSAGE-LOG-DESTINATION.

[Function]
log-message* *log-level format-string &rest format-arguments => result*

Convenience function which calls the message logger of the current acceptor (if there is one) with the same arguments it accepts. Returns NIL if there is no message logger or whatever the message logger returns.

This is the function which Hunchentoot itself uses to log errors it catches during request processing.

[Special variable]
log-lisp-errors-p

Whether Lisp errors in request handlers should be logged.

[Special variable]
log-lisp-backtraces-p

Whether Lisp backtraces should be logged. Only has an effect if *LOG-LISP-ERRORS-P* is true as well.

[Special variable]
log-lisp-warnings-p

Whether Lisp warnings in request handlers should be logged.

[Special variable]
lisp-errors-log-level

Log level for Lisp errors. Should be one of :ERROR (the default), :WARNING, or :INFO.

[Special variable]
lisp-warnings-log-level

Log level for Lisp warnings. Should be one of :ERROR, :WARNING (the default), or :INFO.

Conditions and error handling

This section describes how Hunchentoot deals with exceptional situations. See also the secion about logging.

When an error occurs while processing a request, Hunchentoot's default behavior is to catch the error, log it and optionally display it to the client in the HTML response. This behavior can be customized through the values of a number of special

variables, which are documented below.

[Special variable]
catch-errors-p

If the value of this variable is NIL (the default is T), then errors which happen while a request is handled aren't caught as usual, but instead your Lisp's debugger is invoked. This variable should obviously always be set to a *true* value in a production environment. See MAYBE-INVOKE-DEBUGGER if you want to fine-tune this behaviour.

[Special variable]
show-lisp-errors-p

Whether Lisp errors should be shown in HTML output. Note that this only affects canned responses generated by Lisp. If an error template is present for the "internal server error" status code, this special variable is not used (see acceptor-status-message).

[Special variable]
show-lisp-backtraces-p

Whether Lisp backtraces should be shown in HTML output if *SHOW-LISP-ERRORS-P* is true and an error occurs.

[Generic function]
maybe-invoke-debugger *condition* => |

This generic function is called whenever a condition *condition* is signaled in Hunchentoot. You might want to specialize it on specific condition classes for debugging purposes. The default method invokes the debugger with *condition* if *CATCH-ERRORS-P* is NIL.

[Condition type]
hunchentoot-condition

Superclass for all conditions related to Hunchentoot.

[Condition type]
hunchentoot-error

Superclass for all errors related to Hunchentoot and a subclass of HUNCHENTOOT-CONDITION.

[Condition type]
parameter-error

Signalled if a function was called with incosistent or illegal parameters. A subclass of HUNCHENTOOT-ERROR.

[Condition type]
hunchentoot-warning

Superclass for all warnings related to Hunchentoot and a subclass of HUNCHENTOOT-CONDITION.

Miscellaneous

Various functions and variables which didn't fit into one of the other categories.

[Function]
abort-request-handler *&optional result => result*

This function can be called by a request handler at any time to immediately abort handling the request. This works as if the handler had returned *result*. See the source code of REDIRECT for an example.

[Function]
handle-if-modified-since *time &optional request* => |

This function is designed to be used inside a handler. If the client has sent an 'If-Modified-Since' header (see RFC 2616, section 14.25) and the time specified matches the universal time *time* then the header +HTTP-NOT-MODIFIED+ with no content is immediately returned to the client.

Note that for this function to be useful you should usually send 'Last-Modified' headers back to the client. See the code of CREATE-STATIC-FILE-DISPATCHER-AND-HANDLER for an example.

[Function]
handle-static-file *path &optional content-type => nil*

Sends the file denoted by the pathname designator *path* with content type *content-type* to the client. Sets the necessary handlers. In particular the function employs HANDLE-IF-MODIFIED-SINCE.

If *content-type* is NIL the function tries to determine the correct content type from the file's suffix or falls back to "application/octet-stream" as a last resort.

Note that this function calls SEND-HEADERS internally, so after you've called it, the headers are sent and the return

value of your handler is ignored.

[Function]
redirect *target &key host port protocol add-session-id code* => |

Sends back appropriate headers to redirect the client to *target* (a string).

If *target* is a full URL starting with a scheme, *host*, *port*, and *protocol* are ignored. Otherwise, *target* should denote the path part of a URL, *protocol* must be one of the keywords :HTTP or :HTTPS, and the URL to redirect to will be constructed from *host*, *port*, *protocol*, and *target*.

code must be a 3xx HTTP redirection status code to send to the client. It defaults to 302 ("Found"). If *host* is not provided, the current host (see HOST) will be used. If *protocol* is the keyword :HTTPS, the client will be redirected to a https URL, if it's :HTTP it'll be sent to a http URL. If both *host* and *protocol* aren't provided, then the value of *protocol* will match the current request.

[Function]
require-authorization *&optional realm* => |

Sends back appropriate headers to require basic HTTP authentication (see RFC 2617) for the realm *realm*. The default value for *realm* is "Hunchentoot".

[Function]
no-cache => |

Adds appropriate headers to completely prevent caching on most browsers.

[Function]
ssl-p *&optional acceptor => generalized-boolean*

Whether the current connection to the client is secure. See ACCEPTOR-SSL-P.

[Function]
reason-phrase *return-code => string*

Returns a reason phrase for the HTTP return code *return-code* (which should be an integer) or NIL for return codes Hunchentoot doesn't know.

[Function]
rfc-1123-date *&optional time => string*

Generates a time string according to RFC 1123. Default is current time. This can be used to send a 'Last-Modified' header - see HANDLE-IF-MODIFIED-SINCE.

[Function]
url-encode *string &optional external-format => string*

URL-encodes a string using the external format *external-format*. The default for *external-format* is the value of *HUNCHENTOOT-DEFAULT-EXTERNAL-FORMAT*.

[Function]
url-decode *string &optional external-format => string*

Decodes a URL-encoded string which is assumed to be encoded using the external format *external-format*, i.e. this is the inverse of URL-ENCODE. It is assumed that you'll rarely need this function, if ever. But just in case - here it is. The default for *external-format* is the value of *HUNCHENTOOT-DEFAULT-EXTERNAL-FORMAT*.

[Function]
escape-for-html *string => result*

Escapes the characters #\<, #\>, #\', #\", and #\& for HTML output.

[Function]
http-token-p *object => generalized-boolean*

This function tests whether *object* is a non-empty string which is a *token* according to RFC 2068 (i.e. whether it may be used for, say, cookie names).

[Function]
mime-type *pathspec => result*

Given a pathname designator *pathspec* returns the MIME type (as a string) corresponding to the suffix of the file denoted by *pathspec* (or NIL).

[Function]
within-request-p => *generalized-boolean*

Returns true if in the context of a request. Otherwise, NIL.

[Special variable]
tmp-directory

This should be a pathname denoting a directory where temporary files can be stored. It is used for file uploads.

[Special variable]
header-stream

If this variable is not NIL, it should be bound to a stream to which incoming and outgoing headers will be written for debugging purposes.

[Special variable]
cleanup-function

A designator for a function without arguments which is called on a regular basis if *CLEANUP-INTERVAL* is not NIL. The initial value is the name of a function which invokes a garbage collection on 32-bit versions of LispWorks.

This variable is only available on LispWorks.

[Special variable]
cleanup-interval

Should be NIL or a positive integer. The system calls *CLEANUP-FUNCTION* whenever *CLEANUP-INTERVAL* new worker threads (counted globally across all acceptors) have been created unless the value is NIL. The initial value is 100.

This variable is only available on LispWorks.

Testing

Hunchentoot comes with a test script which verifies that the example web server responds as expected. This test script uses the Drakma HTTP client library and thus shares a significant amount of its base code with Hunchentoot itself. Still, running the test script is a useful confidence test, and it is also possible to run the script across machines in order to verify a new Hunchentoot (or, for that matter Drakma) port.

To run the confidence test, start the example web server. Then, in your Lisp listener, type

```
(hunchentoot-test:test-hunchentoot "http://localhost:4242")
```

You will see some diagnostic output and a summary line that reports whether any tests have failed. (You can also use the example certificate and key files in the test directory and start and test an https server instead.)

[Function]
hunchentoot-test:test-hunchentoot *base-url &key* => |

Runs the built-in confidence test. *base-url* is the base URL to use for testing, it should not have a trailing slash. The keyword arguments accepted are for future extension and should not currently be used.

The script expects the Hunchentoot example test server to be running at the given *base-url* and retrieves various pages from that server, expecting certain responses.

Debugging

By default, Hunchentoot intercepts all errors that occur while executing request handlers, logs them to the log file and displays a static error page to the user. While developing applications, you may want to change that behavior so that the debugger is invoked when an error occurs. You can set the *CATCH-ERRORS-P* to NIL to make that happen. Alternatively, you may want to have Hunchentoot display detailed error information in the error response page. You can set the *SHOW-LISP-ERRORS-P* to a true value to make that happen. If you don't want to see Lisp backtraces in these error pages, you can set *SHOW-LISP-BACKTRACES-P* to NIL.

History

Hunchentoot's predecessor TBNL (which is short for "To Be Named Later") grew over the years as a toolkit that I used for various commercial and private projects. In August 2003, Daniel Barlow started a review of web APIs on the lispweb mailing list and I described the API of my hitherto-unreleased bunch of code (and christened it "TBNL").

It turned out that Jeff Caldwell had worked on something similar so he emailed me and proposed to join our efforts. As I had no immediate plans to release my code (which was poorly organized, undocumented, and mostly CMUCL-specific), I gave it to Jeff and he worked towards a release. He added docstrings, refactored, added some stuff, and based it on KMRCL to make it portable across several Lisp implementations.

Unfortunately, Jeff is at least as busy as I am so he didn't find the time to finish a full release. But in spring 2004 I needed a documented version of the code for a client of mine who thought it would be good if the toolkit were publicly available under an open source license. So I took Jeff's code, refactored again (to sync with the changes I had done in the meantime), and added documentation. This resulted in TBNL 0.1.0 (which initially required mod_lisp as its front-end).

In March 2005, Bob Hutchinson sent patches which enabled TBNL to use other front-ends than mod_lisp. This made me aware that TBNL was already *almost* a full web server, so eventually I wrote Hunchentoot which *was* a full web server, implemented as a

wrapper around TBNL. Hunchentoot 0.1.0 was released at the end of 2005 and was originally LispWorks-only.

Hunchentoot 0.4.0, released in October 2006, was the first release which also worked with other Common Lisp implementations. It is a major rewrite and also incorporates most of TBNL and replaces it completely.

Hunchentoot 1.0.0, released in February 2009, is again a major rewrite and should be considered work in progress. It moved to using the usocket and Bordeaux Threads libraries for non-LispWorks Lisps, thereby removing most of the platform dependent code. Threading behaviour was made controllable through the introduction of taskmasters. mod_lisp support and several other things were removed in this release to simplify the code base (and partly due to the lack of interest). Several architectural changes (lots of them not backwards-compatible) were made to ease customization of Hunchentoot's behaviour. A significant part of the 1.0.0 redesign was done by Hans Hübner.

Symbol index

Here are all exported symbols of the HUNCHENTOOT package in alphabetical order linked to their corresponding documentation entries:

- *acceptor* *Special variable*
- *catch-errors-p* *Special variable*
- *cleanup-function* *Special variable*
- *cleanup-interval* *Special variable*
- *content-types-for-url-rewrite* *Special variable*
- *default-connection-timeout* *Special variable*
- *default-content-type* *Special variable*
- *dispatch-table* *Special variable*
- *file-upload-hook* *Special variable*
- *header-stream* *Special variable*
- *hunchentoot-default-external-format* *Special variable*
- *lisp-errors-log-level* *Special variable*
- *lisp-warnings-log-level* *Special variable*
- *log-lisp-backtraces-p* *Special variable*
- *log-lisp-errors-p* *Special variable*
- *log-lisp-warnings-p* *Special variable*
- *methods-for-post-parameters* *Special variable*
- *reply* *Special variable*
- *request* *Special variable*
- *rewrite-for-session-urls* *Special variable*
- *session* *Special variable*
- *session-gc-frequency* *Special variable*
- *session-max-time* *Special variable*
- *session-secret* *Special variable*
- *show-lisp-backtraces-p* *Special variable*
- *show-lisp-errors-p* *Special variable*
- *tmp-directory* *Special variable*
- *use-remote-addr-for-sessions* *Special variable*
- *use-user-agent-for-sessions* *Special variable*
- +http-accepted+ *Constant*
- +http-authorization-required+ *Constant*
- +http-bad-gateway+ *Constant*
- +http-bad-request+ *Constant*
- +http-conflict+ *Constant*
- +http-continue+ *Constant*
- +http-created+ *Constant*
- +http-expectation-failed+ *Constant*
- +http-failed-dependency+ *Constant*
- +http-forbidden+ *Constant*
- +http-gateway-time-out+ *Constant*
- +http-gone+ *Constant*
- +http-internal-server-error+ *Constant*
- +http-length-required+ *Constant*
- +http-method-not-allowed+ *Constant*
- +http-moved-permanently+ *Constant*
- +http-moved-temporarily+ *Constant*
- +http-multi-status+ *Constant*
- +http-multiple-choices+ *Constant*
- +http-no-content+ *Constant*
- +http-non-authoritative-information+ *Constant*
- +http-not-acceptable+ *Constant*
- +http-not-found+ *Constant*
- +http-not-implemented+ *Constant*
- +http-not-modified+ *Constant*
- +http-ok+ *Constant*
- +http-partial-content+ *Constant*
- +http-payment-required+ *Constant*
- +http-precondition-failed+ *Constant*

- +http-proxy-authentication-required+ *Constant*
- +http-request-entity-too-large+ *Constant*
- +http-request-time-out+ *Constant*
- +http-request-uri-too-large+ *Constant*
- +http-requested-range-not-satisfiable+ *Constant*
- +http-reset-content+ *Constant*
- +http-see-other+ *Constant*
- +http-service-unavailable+ *Constant*
- +http-switching-protocols+ *Constant*
- +http-temporary-redirect+ *Constant*
- +http-unsupported-media-type+ *Constant*
- +http-use-proxy+ *Constant*
- +http-version-not-supported+ *Constant*
- abort-request-handler *Function*
- accept-connections *Generic function*
- acceptor *Standard class*
- acceptor-access-log-destination *Generic accessor*
- acceptor-address *Generic reader*
- acceptor-dispatch-request *Generic function*
- acceptor-document-root *Generic accessor*
- acceptor-error-template-directory *Generic accessor*
- acceptor-input-chunking-p *Generic accessor*
- acceptor-listen-backlog *Generic function*
- acceptor-log-access *Generic function*
- acceptor-log-message *Generic function*
- acceptor-message-log-destination *Generic accessor*
- acceptor-name *Generic accessor*
- acceptor-output-chunking-p *Generic accessor*
- acceptor-persistent-connections-p *Generic accessor*
- acceptor-port *Generic reader*
- acceptor-read-timeout *Generic reader*
- acceptor-remove-session *Generic function*
- acceptor-reply-class *Generic accessor*
- acceptor-request-class *Generic accessor*
- acceptor-ssl-certificate-file *Generic reader*
- acceptor-ssl-p *Generic function*
- acceptor-ssl-privatekey-file *Generic reader*
- acceptor-ssl-privatekey-password *Generic reader*
- acceptor-status-message *Generic function*
- acceptor-write-timeout *Generic reader*
- authorization *Function*
- aux-request-value *Accessor*
- content-length *Generic reader*
- content-length* *Accessor*
- content-type *Generic reader*
- content-type* *Accessor*
- cookie-in *Function*
- cookie-out *Function*
- cookies-in *Generic reader*
- cookies-in* *Function*
- cookies-out *Generic accessor*
- cookies-out* *Accessor*
- create-folder-dispatcher-and-handler *Function*
- create-prefix-dispatcher *Function*
- create-regex-dispatcher *Function*
- create-request-handler-thread *Generic function*
- create-static-file-dispatcher-and-handler *Function*
- define-easy-handler *Macro*
- delete-aux-request-value *Function*
- delete-session-value *Function*
- detach-socket *Generic function*
- dispatch-easy-handlers *Function*
- easy-acceptor *Standard class*
- easy-ssl-acceptor *Standard class*
- escape-for-html *Function*
- execute-acceptor *Generic function*
- get-parameter *Function*
- get-parameters *Generic reader*
- get-parameters* *Function*
- handle-if-modified-since *Function*
- handle-incoming-connection *Generic function*
- handle-request *Generic function*
- handle-static-file *Function*
- header-in *Generic reader*
- header-in* *Function*

- header-out *Accessor*
- headers-in *Generic reader*
- headers-in* *Function*
- headers-out *Generic reader*
- headers-out* *Function*
- host *Function*
- http-token-p *Function*
- hunchentoot-condition *Condition type*
- hunchentoot-error *Condition type*
- hunchentoot-test:test-hunchentoot *Function*
- hunchentoot-warning *Condition type*
- initialize-connection-stream *Generic function*
- local-addr *Generic reader*
- local-addr* *Function*
- local-port *Generic reader*
- local-port* *Function*
- log-message* *Function*
- maybe-invoke-debugger *Generic function*
- mime-type *Function*
- multi-threaded-taskmaster *Standard class*
- next-session-id *Generic function*
- no-cache *Function*
- one-thread-per-connection-taskmaster *Standard class*
- parameter *Function*
- parameter-error *Condition type*
- post-parameter *Function*
- post-parameters *Generic reader*
- post-parameters* *Function*
- process-connection *Generic function*
- process-request *Generic function*
- query-string *Generic reader*
- query-string* *Function*
- raw-post-data *Function*
- real-remote-addr *Function*
- reason-phrase *Function*
- recompute-request-parameters *Function*
- redirect *Function*
- referer *Function*
- regenerate-session-cookie-value *Function*
- remote-addr *Generic reader*
- remote-addr* *Function*
- remote-port *Generic reader*
- remote-port* *Function*
- remove-session *Function*
- reply *Standard class*
- reply-external-format *Generic accessor*
- reply-external-format* *Accessor*
- request *Standard class*
- request-acceptor *Generic reader*
- request-method *Generic reader*
- request-method* *Function*
- request-uri *Generic reader*
- request-uri* *Function*
- require-authorization *Function*
- reset-connection-stream *Generic function*
- reset-session-secret *Function*
- reset-sessions *Function*
- return-code *Generic accessor*
- return-code* *Accessor*
- rfc-1123-date *Function*
- script-name *Generic reader*
- script-name* *Function*
- send-headers *Function*
- server-protocol *Generic reader*
- server-protocol* *Function*
- session *Standard class*
- session-cookie-name *Generic function*
- session-cookie-value *Generic function*
- session-created *Generic function*
- session-db *Generic accessor*
- session-db-lock *Generic function*
- session-gc *Function*
- session-id *Generic function*
- session-max-time *Generic accessor*
- session-remote-addr *Generic function*

- session-start *Generic function*
- session-too-old-p *Function*
- session-user-agent *Generic function*
- session-value *Accessor*
- session-verify *Generic function*
- set-cookie *Function*
- set-cookie* *Function*
- shutdown *Generic function*
- single-threaded-taskmaster *Standard class*
- ssl-acceptor *Standard class*
- ssl-p *Function*
- start *Generic function*
- start-listening *Generic function*
- start-session *Function*
- start-thread *Generic function*
- stop *Generic function*
- taskmaster *Standard class*
- taskmaster-acceptor *Generic accessor*
- url-decode *Function*
- url-encode *Function*
- user-agent *Function*
- within-request-p *Function*

Acknowledgements

Thanks to Jeff Caldwell - TBNL would not have been released without his efforts. Thanks to Stefan Scholl and Travis Cross for various additions and fixes to TBNL, to Michael Weber for initial file upload code, and to Janis Dzerins for his RFC 2388 code. Thanks to Bob Hutchison for his code for multiple front-ends (which made me realize that TBNL was already pretty close to a "real" web server) and the initial UTF-8 example. Thanks to Hans Hübner for a lot of architectural and implementation enhancements for the 1.0.0 release and also for transferring the documentation to sane XHTML. Thanks to John Foderaro's AllegroServe for inspiration. Thanks to Uwe von Loh for the Hunchentoot logo.

Hunchentoot originally used code from ACL-COMPAT, specifically the chunking code from Jochen Schmidt. (This has been replaced by Chunga.) When I ported Hunchentoot to other Lisps than LispWorks, I stole code from ACL-COMPAT, KMRCL, and trivial-sockets for implementation-dependent stuff like sockets and MP. (This has been replaced by Bordeaux Threads and usocket.)

Parts of this documentation were prepared with DOCUMENTATION-TEMPLATE, no animals were harmed.

BACK TO MY HOMEPAGE