Web Server: HTTP Server

Version 6.1

Jay McCarthy

August 1, 2014

This manual describes the internals of the Racket Web Server.

1 Dispatching Server

The Web Server is just a configuration of a dispatching server.

1.1 Dispatching Server Signatures

```
(require web-server/private/dispatch-server-sig)
```
package: web-server-lib

The web-server/private/dispatch-server-sig library provides two signatures.

dispatch-server^ : signature

The dispatch-server^ signature is an alias for web-server^.

```
(serve)
 → (->* () (#:confirmation-channel (or/c false/c async-channel?)) (-> void))
```

Runs the server—the confirmation channel will be send an exception if one occurs starting the server or the port number if there is none—and returns a procedure that shuts down the server.

```
(serve-ports ip op) → void
  ip : input-port?
  op : output-port?
```

Serves a single connection represented by the ports *ip* and *op*.

dispatch-server-config^ : signature

port : tcp-listen-port?

Specifies the port to serve on.

listen-ip : (or/c string? false/c)

Passed to tcp-listen.

max-waiting : exact-nonnegative-integer?

Passed to tcp-listen.

```
initial-connection-timeout : integer?
```

Specifies the initial timeout given to a connection.

```
(read-request c p port-addresses) → any/c boolean?
  c : connection?
  p : tcp-listen-port?
  port-addresses : (input-port? . -> . (values string? string?))
```

Defines the way the server reads requests off connections to be passed to dispatch.

```
dispatch : (-> connection? any/c void)
```

How to handle requests.

1.2 Dispatching Server Unit

```
(require web-server/private/dispatch-server-unit)
```
package: web-server-lib

The web-server/private/dispatch-server-unit module provides the unit that actually implements a dispatching server.

```
dispatch-server@ : (unit/c (import tcp^ dispatch-server-config^)
                           (export dispatch-server^))
```

Runs the dispatching server config in a very basic way, except that it uses §5.2 “Connection Manager” to manage connections.

1.3 Threads and Custodians

The dispatching server runs in a dedicated thread. Every time a connection is initiated, a new thread is started to handle it. Connection threads are created inside a dedicated custodian that is a child of the server’s custodian. When the server is used to provide servlets, each servlet also receives a new custodian that is a child of the server’s custodian **not** the connection custodian.

2 Dispatchers

Since the Web Server is really just a particular configuration of a dispatching server, there are several dispatchers that are defined to support the Web Server. Other dispatching servers may find these useful. In particular, if you want a peculiar processing pipeline for your Web Server installation, refer to this documentation.

2.1 General

```
(require web-server/dispatchers/dispatch)
                        package: web-server-lib
```

This module provides a few functions for dispatchers in general.

```
dispatcher/c : contract?
```

Equivalent to `(-> connection? request? void)`.

```
(dispatcher-interface-version/c any) → boolean?
  any : any/c
```

Equivalent to `(symbols 'v1)`

```
(struct exn:dispatcher ()
    #:extra-constructor-name make-exn:dispatcher)
```

An exception thrown to indicate that a dispatcher does not apply to a particular request.

```
(next-dispatcher) → void
```

Raises a `exn:dispatcher`

As the `dispatcher/c` contract suggests, a dispatcher is a function that takes a connection and request object and does something to them. Mostly likely it will generate some response and output it on the connection, but it may do something different. For example, it may apply some test to the request object, perhaps checking for a valid source IP address, and error if the test is not passed, and call `next-dispatcher` otherwise.

Consider the following example dispatcher, that captures the essence of URL rewriting:

```
; (url? -> url?) dispatcher/c -> dispatcher/c
(lambda (rule inner)
  (lambda (conn req)
    ; Call the inner dispatcher...
    (inner conn
           ; with a new request object...
           (struct-copy request req
                        ; with a new URL!
                        [request-uri (rule (request-uri req))]))))
```

2.2 Mapping URLs to Paths

```
(require web-server/dispatchers/filesystem-map)
```
package: web-server-lib

This module provides a means of mapping URLs to paths on the filesystem.

```
url->path/c : contract?
```

This contract is equivalent to `(->* (url?) (path? (listof path-piece?)))`. The returned `path?` is the path on disk. The list is the list of path elements that correspond to the path of the URL.

```
(make-url->path base) → url->path/c
  base : path-string?
```

The `url->path/c` returned by this procedure considers the root URL to be *base*. It ensures that `".."`s in the URL do not escape the *base* and removes them silently otherwise.

```
(make-url->valid-path url->path) → url->path/c
  url->path : url->path/c
```

Runs the underlying *url->path*, but only returns if the path refers to a file that actually exists. If it is does not, then the suffix elements of the URL are removed until a file is found. If this never occurs, then an error is thrown.

This is primarily useful for dispatchers that allow path information after the name of a service to be used for data, but where the service is represented by a file. The most prominent example is obviously servlets.

```
(filter-url->path regex url->path) → url->path/c
  regex : regexp?
  url->path : url->path/c
```

Runs the underlying *url->path* but will only return if the path, when considered as a string, matches the *regex*. This is useful to disallow strange files, like GIFs, from being considered servlets when using the servlet dispatchers. It will return a `exn:fail:filesystem:exists?` exception if the path does not match.

2.3 Sequencing

```
(require web-server/dispatchers/dispatch-sequencer)
```
package: `web-server-lib`

The `web-server/dispatchers/dispatch-sequencer` module defines a dispatcher constructor that invokes a sequence of dispatchers until one applies.

```
(make dispatcher ...) → dispatcher/c
  dispatcher : dispatcher/c
```

Invokes each *dispatcher*, invoking the next if the first calls `next-dispatcher`. If no *dispatcher* applies, then it calls `next-dispatcher` itself.

2.4 Timeouts

```
(require web-server/dispatchers/dispatch-timeout)
```
package: `web-server-lib`

The `web-server/dispatchers/dispatch-timeout` module defines a dispatcher constructor that changes the timeout on the connection and calls the next dispatcher.

```
(make new-timeout) → dispatcher/c
  new-timeout : integer?
```

Changes the timeout on the connection with `adjust-connection-timeout!` called with *new-timeout*.

2.5 Lifting Procedures

```
(require web-server/dispatchers/dispatch-lift)
```

package: web-server-lib

The web-server/dispatchers/dispatch-lift module defines a dispatcher constructor.

```
(make proc) → dispatcher/c
  proc : (request? . -> . response?)
```

Constructs a dispatcher that calls *proc* on the request object, and outputs the response to the connection.

2.6 Filtering Requests

```
(require web-server/dispatchers/dispatch-filter)
```
package: web-server-lib

The web-server/dispatchers/dispatch-filter module defines a dispatcher constructor that calls an underlying dispatcher with all requests that pass a predicate.

```
(make regex inner) → dispatcher/c
  regex : regexp?
  inner : dispatcher/c
```

Calls *inner* if the URL path of the request, converted to a string, matches *regex*. Otherwise, calls next-dispatcher.

2.7 Procedure Invocation upon Request

```
(require web-server/dispatchers/dispatch-pathprocedure)
```
package: web-server-lib

The web-server/dispatchers/dispatch-pathprocedure module defines a dispatcher constructor for invoking a particular procedure when a request is given to a particular URL path.

```
(make path proc) → dispatcher/c
  path : string?
  proc : (request? . -> . response?)
```

Checks if the request URL path as a string is equal to *path* and if so, calls *proc* for a response.

This is used in the standard Web Server pipeline to provide a URL that refreshes the password file, servlet cache, etc.

2.8 Logging

```
(require web-server/dispatchers/dispatch-log)
```
package: web-server-lib

The web-server/dispatchers/dispatch-log module defines a dispatcher constructor for transparent logging of requests.

format-req/c : contract?

Equivalent to (-> request? string?).

paren-format : format-req/c

Formats a request by:

```
(format
 "~s\n"
 (list 'from (request-client-ip req)
       'to (request-host-ip req)
       'for (url->string (request-uri req)) 'at
       (date->string
        (seconds->date (current-seconds)) #t)))
```

extended-format : format-req/c

Formats a request by:

```
(format
 "~s\n"
 `((client-ip ,(request-client-ip req))
   (host-ip ,(request-host-ip req))
   (referer
    ,(let ([R (headers-assq*
               #"Referer"
               (request-headers/raw req))])
       (if R
           (header-value R)
```

```
          #f)))
    (uri ,(url->string (request-uri req)))
    (time ,(current-seconds))))
```

```
apache-default-format : format-req/c
```

Formats a request like Apache's default. However, Apache's default includes information about the response to a request, which this function does not have access to, so it defaults the last two fields to 200 and 512.

```
log-format/c : contract?
```

Equivalent to `(symbols 'parenthesized-default 'extended 'apache-default)`.

```
(log-format->format id) → format-req/c
  id : log-format/c
```

Maps `'parenthesized-default` to `paren-format`, `'extended` to `extended-format`, and `'apache-default` to `apache-default-format`.

```
(make [#:format format #:log-path log-path]) → dispatcher/c
  format : format-req/c = paren-format
  log-path : path-string? = "log"
```

Logs requests to *log-path* by using *format* to format the requests. Then invokes `next-dispatcher`.

2.9 Password Protection

```
(require web-server/dispatchers/dispatch-passwords)
```
package: web-server-lib

The `web-server/dispatchers/dispatch-passwords` module defines a dispatcher constructor that performs HTTP Basic authentication filtering.

```
denied?/c : contract?
```

Equivalent to `(-> request? (or/c false/c string?))`. The return is the authentication realm as a string if the request is not authorized and `#f` if the request *is* authorized.

```
(make  denied?
       [#:authentication-responder authentication-responder])
 → dispatcher/c
  denied? : denied?/c
  authentication-responder : (url? header? . -> . response?)
                           = (gen-authentication-responder "forbidden.html")
```

A dispatcher that checks if the request is denied based on *denied?*. If so, then *authentication-responder* is called with a header that requests credentials. If not, then next-dispatcher is invoked.

```
authorized?/c : contract?
```

Equivalent to (-> string? (or/c false/c bytes?) (or/c false/c bytes?) (or/c false/c string?)). The input is the URI as a string and the username and passwords as bytes. The return is the authentication realm as a string if the user is not authorized and #f if the request *is* authorized.

```
(make-basic-denied?/path authorized?) → denied?/c
  authorized? : authorized?/c
```

Creates a denied procedure from an authorized procedure.

```
(password-file->authorized? password-file)
 → (-> void) authorized?/c
  password-file : path-string?
```

Creates an authorization procedure based on the given password file. The first returned value is a procedure that refreshes the password cache used by the authorization procedure.

password-file is parsed as:

```
(list ([domain : string?]
       [path : string?] ; This string is interpreted as a regex
       (list [user : symbol?]
             [pass : string?])
       ...)
      ...)
```

For example:

```
'(("secret stuff" "/secret(/.*)?" (bubba "bbq") (Billy "BoB")))
```

2.10 Virtual Hosts

```
(require web-server/dispatchers/dispatch-host)
```

package: web-server-lib

The web-server/dispatchers/dispatch-host module defines a dispatcher constructor that calls a different dispatcher based upon the host requested.

```
(make lookup-dispatcher) → dispatcher/c
  lookup-dispatcher : (symbol? . -> . dispatcher/c)
```

Extracts a host from the URL requested, or the Host HTTP header, calls *lookup-dispatcher* with the host, and invokes the returned dispatcher. If no host can be extracted, then 'none is used.

2.11 Serving Files

```
(require web-server/dispatchers/dispatch-files)
                                  package: web-server-lib
```

The web-server/dispatchers/dispatch-files module allows files to be served. It defines a dispatcher construction procedure.

```
(make  #:url->path url->path
       [#:path->mime-type path->mime-type
        #:indices indices])                → dispatcher/c
  url->path : url->path/c
  path->mime-type : (path? . -> . (or/c false/c bytes)?)
                  = (lambda (path) #f)
  indices : (listof string?) = (list "index.html" "index.htm")
```

Uses *url->path* to extract a path from the URL in the request object. If this path does not exist, then the dispatcher does not apply and next-dispatcher is invoked. If the path is a directory, then the *indices* are checked in order for an index file to serve. In that case, or in the case of a path that is a file already, *path->mime-type* is consulted for the MIME Type of the path. The file is then streamed out the connection object.

This dispatcher supports HTTP Range GET requests and HEAD requests.

2.12 Serving Servlets

```
(require web-server/dispatchers/dispatch-servlets)
                                    package: web-server-lib
```

The web-server/dispatchers/dispatch-servlets module defines a dispatcher constructor that runs servlets.

```
url->servlet/c : contract?
```

Equivalent to `(-> url? servlet?)`

```
(make-cached-url->servlet url->path
                          path->serlvet)
 → (-> void) url->servlet/c
  url->path : url->path/c
  path->serlvet : path->servlet/c
```

The first return value flushes the cache. The second is a procedure that uses *url->path* to resolve the URL to a path, then uses path->servlet to resolve that path to a servlet, caching the results in an internal table.

```
(make  url->servlet
       [#:responders-servlet-loading responders-servlet-loading
        #:responders-servlet responders-servlet])
 → dispatcher/c
  url->servlet : url->servlet/c
  responders-servlet-loading : (url? exn? . -> . can-be-response?)
                             = servlet-loading-responder
  responders-servlet : (url? exn? . -> . can-be-response?)
                     = servlet-error-responder
```

This dispatcher runs racket servlets, using *url->servlet* to resolve URLs to the underlying servlets. If servlets have errors loading, then *responders-servlet-loading* is used. Other errors are handled with *responders-servlet*. If a servlet raises calls next-dispatcher, then the signal is propagated by this dispatcher.

2.12.1 Setting Up Servlets

```
(require web-server/servlet/setup)
```
package: web-server-lib

This module is used internally to build and load servlets. It may be useful to those who are trying to extend the server.

```
(make-v1.servlet directory timeout start) → servlet?
  directory : path-string?
  timeout : integer?
  start : (request? . -> . can-be-response?)
```

Creates a version 1 servlet that uses *directory* as its current directory, a timeout manager with a *timeout* timeout, and *start* as the request handler.

```
(make-v2.servlet directory manager start) → servlet?
  directory : path-string?
  manager : manager?
  start : (request? . -> . can-be-response?)
```

Creates a version 2 servlet that uses *directory* as its current directory, a *manager* as the continuation manager, and *start* as the request handler.

```
(make-stateless.servlet directory
                        stuffer
                        manager
                        start)    → servlet?
  directory : path-string?
  stuffer : (stuffer/c serializable? bytes?)
  manager : manager?
  start : (request? . -> . can-be-response?)
```

Creates a stateless web-server servlet that uses *directory* as its current directory, *stuffer* as its stuffer, and *manager* as the continuation manager, and *start* as the request handler.

```
default-module-specs : (listof module-path?)
```

The modules that the Web Server needs to share with all servlets.

```
path->servlet/c : contract?
```

Equivalent to (-> path? servlet?).

```
 (make-default-path->servlet
  [#:make-servlet-namespace make-servlet-namespace
   #:timeouts-default-servlet timeouts-default-servlet])
 → path->servlet/c
  make-servlet-namespace : make-servlet-namespace/c
                         = (make-make-servlet-namespace)
  timeouts-default-servlet : integer? = 30
```

Constructs a procedure that loads a servlet from the path in a namespace created with *make-servlet-namespace*, using a timeout manager with *timeouts-default-servlet* as the default timeout (if no manager is given.)

2.12.2 Servlet Namespaces

```
(require web-server/configuration/namespace)
```
package: web-server-lib

This module provides a function to help create the `make-servlet-namespace` procedure needed by the `make` function of `web-server/dispatchers/dispatch-servlets`.

```
make-servlet-namespace/c : contract?
```

Equivalent to

```
(->* ()
     (#:additional-specs (listof module-path?))
     namespace?)
```

```
(make-make-servlet-namespace #:to-be-copied-module-specs to-be-
copied-module-specs)
 → make-servlet-namespace/c
  to-be-copied-module-specs : (listof module-path?)
```

This function creates a function that when called will construct a new `namespace` that has all the modules from *`to-be-copied-module-specs`* and `additional-specs`, as well as `racket` and `mred`, provided they are already attached to the `(current-namespace)` of the call-site.

Example:

```
(make-make-servlet-namespace
 #:to-be-copied-module-specs '((lib "database.rkt" "my-module")))
```

Why this is useful

A different namespace is needed for each servlet, so that if servlet A and servlet B both use a stateful module C, they will be isolated from one another. We see the Web Server as an operating system for servlets, so we inherit the isolation requirement on operating systems.

However, there are some modules which must be shared. If they were not, then structures cannot be passed from the Web Server to the servlets, because Racket's structures are generative.

Since, on occasion, a user will actually wanted servlets A and B to interact through module C. A custom `make-servlet-namespace` can be created, through this procedure, that

attaches module C to all servlet namespaces. Through other means (see §2 “Dispatchers”) different sets of servlets can share different sets of modules.

2.12.3 Internal Servlet Representation

```
(require web-server/private/servlet)
                    package: web-server-lib
```

```
(struct servlet (custodian namespace manager directory handler)
    #:extra-constructor-name make-servlet
    #:mutable)
  custodian : custodian?
  namespace : namespace?
  manager : manager?
  directory : path-string?
  handler : (request? . -> . can-be-response?)
```

Instances of this structure hold the necessary parts of a servlet: the `custodian` responsible for the servlet’s resources, the `namespace` the servlet is executed within, the `manager` responsible for the servlet’s continuations, the current `directory` of the servlet, and the `handler` for all requests to the servlet.

2.13 Statistics

```
(require web-server/dispatchers/dispatch-stat)
                              package: web-server-lib
```

The `web-server/dispatchers/dispatch-stat` module provides services related to performance statistics.

```
(make-gc-thread time) → thread?
  time : integer?
```

Starts a thread that calls `(collect-garbage)` every *time* seconds.

```
(make) → dispatcher/c
```

Returns a dispatcher that prints memory usage on every request.

2.14 Limiting Requests

```
(require web-server/dispatchers/limit)
                    package: web-server-lib
```

The `web-server/dispatchers/limit` module provides a wrapper dispatcher that limits how many requests are serviced at once.

```
(make limit inner [#:over-limit over-limit]) → dispatcher/c
  limit : number?
  inner : dispatcher/c
  over-limit : (symbols 'block 'kill-new 'kill-old) = 'block
```

Returns a dispatcher that defers to *inner* for work, but will forward a maximum of *limit* requests concurrently.

If there are no additional spaces inside the limit and a new request is received, the *over-limit* option determines what is done. The default (`'block`) causes the new request to block until an old request is finished being handled. If *over-limit* is `'kill-new`, then the new request handler is killed—a form of load-shedding. If *over-limit* is `'kill-old`, then the oldest request handler is killed—prioritizing new connections over old. (This setting is a little dangerous because requests might never finish if there is constant load.)

Consider this example:

```
#lang racket

(require web-server/web-server
         web-server/http
         web-server/http/response
         (prefix-in limit: web-server/dispatchers/limit)
         (prefix-in filter: web-server/dispatchers/dispatch-
filter)
         (prefix-in sequencer: web-server/dispatchers/dispatch-
sequencer))

(serve #:dispatch
       (sequencer:make
        (filter:make
         #rx"/limited"
         (limit:make
          5
          (lambda (conn req)
           (output-response/method
            conn
```

```
            (response/full
             200 #"Okay"
             (current-seconds) TEXT/HTML-MIME-TYPE
             empty
             (list (string->bytes/utf-8
                    (format "hello world ~a"
                            (sort (build-list 100000 (λ x (random 1000)))
                                  <)))))
            (request-method req)))
          #:over-limit 'block))
        (lambda (conn req)
          (output-response/method
           conn
           (response/full 200 #"Okay"
                          (current-seconds) TEXT/HTML-MIME-TYPE
                          empty
                          (list #"<html><body>Unlimited</body></html>"))
           (request-method req))))
      #:port 8080)

(do-not-return)
```

3 Launching Servers

```
(require web-server/web-server)
```
package: web-server-lib

This module provides functions for launching dispatching servers.

```
(serve
 #:dispatch dispatch
 [#:confirmation-channel confirmation-channel
  #:connection-close? connection-close?
  #:tcp@ tcp@
  #:port port
  #:listen-ip listen-ip
  #:max-waiting max-waiting
  #:initial-connection-timeout initial-connection-timeout])
→ (-> void)
 dispatch : dispatcher/c
 confirmation-channel : (or/c false/c async-channel?) = #f
 connection-close? : boolean? = #f
 tcp@ : (unit/c (import) (export tcp^)) = raw:tcp@
 port : tcp-listen-port? = 80
 listen-ip : (or/c string? false/c) = #f
 max-waiting : integer? = 511
 initial-connection-timeout : integer? = 60
```

Constructs an appropriate `dispatch-server-config^`, invokes the `dispatch-server@`, and calls its `serve` function.

If *connection-close?* is `#t`, then every connection is closed after one request. Otherwise, the client decides based on what HTTP version it uses.

The `#:tcp@` keyword is provided for building an SSL server. See §6.3 “How do I set up the server to use HTTPS?”.

Here’s an example of a simple web server that serves files from a given path:

```
(define (start-file-server base)
  (serve
   #:dispatch
   (files:make
    #:url->path (make-url->path base)
    #:path->mime-type
    (lambda (path)
      #"application/octet-stream"))
   #:port 8080))
```

```
(serve/ports
 #:dispatch dispatch
 [#:confirmation-channel confirmation-channel
  #:connection-close? connection-close?
  #:tcp@ tcp@
  #:ports ports
  #:listen-ip listen-ip
  #:max-waiting max-waiting
  #:initial-connection-timeout initial-connection-timeout])
→ (-> void)
 dispatch : dispatcher/c
 confirmation-channel : (or/c false/c async-channel?) = #f
 connection-close? : boolean? = #f
 tcp@ : (unit/c (import) (export tcp^)) = raw:tcp@
 ports : (listof tcp-listen-port?) = (list 80)
 listen-ip : (or/c string? false/c) = #f
 max-waiting : integer? = 511
 initial-connection-timeout : integer? = 60
```

Calls `serve` multiple times, once for each `port`, and returns a function that shuts down all of the server instances.

```
(serve/ips+ports
 #:dispatch dispatch
 [#:confirmation-channel confirmation-channel
  #:connection-close? connection-close?
  #:tcp@ tcp@
  #:ips+ports ips+ports
  #:max-waiting max-waiting
  #:initial-connection-timeout initial-connection-timeout])
→ (-> void)
 dispatch : dispatcher/c
 confirmation-channel : (or/c false/c async-channel?) = #f
 connection-close? : boolean? = #f
 tcp@ : (unit/c (import) (export tcp^)) = raw:tcp@
 ips+ports : (listof (cons/c (or/c string? false/c) (listof tcp-listen-port?)))
           = (list (cons #f (list 80)))
 max-waiting : integer? = 511
 initial-connection-timeout : integer? = 60
```

Calls `serve/ports` multiple times, once for each `ip`, and returns a function that shuts down all of the server instances.

```
(serve/web-config@ config@ [#:tcp@ tcp@]) → (-> void)
  config@ : (unit/c (import) (export web-config^))
  tcp@ : (unit/c (import) (export tcp^)) = raw:tcp@
```

Starts the Web Server with the settings defined by the given `web-config^` unit.

It is very useful to combine this with `configuration-table->web-config@` and `configuration-table-sexpr->web-config@`:

```
(serve/web-config@
 (configuration-table->web-config@
  default-configuration-table-path))
```

```
(do-not-return) → void
```

This function does not return. If you are writing a script to load the Web Server you are likely to want to call this functions at the end of your script.

3.1 Simple Single Servlet Servers

```
(require web-server/servlet-dispatch)
```
package: `web-server-lib`

These functions optimize the construction of dispatchers and launching of servers for single servlets and interactive development.

```
(dispatch/servlet
 start
 [#:regexp regexp
  #:stateless? stateless?
  #:stuffer stuffer
  #:manager manager
  #:current-directory servlet-current-directory
  #:responders-servlet-loading responders-servlet-loading
  #:responders-servlet responders-servlet])
→ dispatcher/c
 start : (request? . -> . response?)
 regexp : regexp? = #rx""
 stateless? : boolean? = #f
 stuffer : (stuffer/c serializable? bytes?) = default-stuffer
 manager : manager?
         = (make-threshold-LRU-manager #f (* 1024 1024 64))
```

```
  servlet-current-directory : path-string? = (current-directory)
  responders-servlet-loading : (url? any/c . -> . can-be-response?)
                             = servlet-loading-responder
  responders-servlet : (url? any/c . -> . can-be-response?)
                     = servlet-error-responder
```

serve/servlet starts a server and uses a particular dispatching sequence. For some applications, this nails down too much, but users are conflicted, because the interface is so convenient. For those users, dispatch/servlet does the hardest part of serve/servlet and constructs a dispatcher just for the *start* servlet.

The dispatcher responds to requests that match *regexp*. The current directory of servlet execution is *servlet-current-directory*.

If *stateless?* is true, then the servlet is run as a stateless

```
#lang web-server
```

module and *stuffer* is used as the stuffer.

The servlet is loaded with *manager* as its continuation manager. (The default manager limits the amount of memory to 64 MB and deals with memory pressure as discussed in the make-threshold-LRU-manager documentation.)

The servlet is run in the (current-namespace).

If a servlet fails to load, *responders-servlet-loading* is used. If a servlet errors during its operation, *responders-servlet* is used.

```
(serve/launch/wait  make-dispatcher
                   [#:connection-close? connection-close?
                    #:launch-path launch-path
                    #:banner? banner?
                    #:listen-ip listen-ip
                    #:port port
                    #:max-waiting max-waiting
                    #:ssl-cert ssl-cert
                    #:ssl-key ssl-key])
 → void
  make-dispatcher : (semaphore? . -> . dispatcher/c)
  connection-close? : boolean? = #f
  launch-path : (or/c false/c string?) = #f
  banner? : boolean? = #f
  listen-ip : (or/c false/c string?) = "127.0.0.1"
  port : number? = 8000
```

```
max-waiting : exact-nonnegative-integer? = 511
ssl-cert : (or/c false/c path-string?) = #f
ssl-key : (or/c false/c path-string?) = #f
```

The other interesting part of `serve/servlet` is its ability to start up a server and immediately launch a browser at it. This is provided by `serve/launch/wait`.

It starts a server using the result of *make-dispatcher* as the dispatcher. *make-dispatcher* is supplied a semaphore that if posted, will cause the server to quit.

If *launch-path* is not false, then a browser is launched with that path appended to the URL to the server itself.

If *banner?* is true, then a banner is printed informing the user of the server's URL.

The server listens on *listen-ip* and port *port*. If *listen-ip* is `#f`, then the server accepts connections to all of the listening machine's addresses. Otherwise, the server accepts connections only at the interface(s) associated with the given string. For example, providing `"127.0.0.1"` (the default) as *listen-ip* creates a server that accepts only connections to `"127.0.0.1"` (the loopback interface) from the local machine.

max-waiting is passed to `serve` to control the TCP backlog.

If *ssl-key* and *ssl-cert* are not false, then the server runs in HTTPS mode with *ssl-cert* and *ssl-key* as paths to the certificate and private key.

If *connection-close?* is `#t`, then every connection is closed after one request. Otherwise, the client decides based on what HTTP version it uses.

4 Web Servers

A Web server is a unit with the `web-server^` signature. The most common way to construct one is to provide a `web-config^` unit to the `web-server@` unit. The most common way to construct a `web-config^` unit is to use `configuration-table->web-config@` to produce one from a configuration table file, such as the one that is shipped with Racket in `default-configuration-table-path`.

4.1 Server Units

4.1.1 Signature

```
(require web-server/web-server-sig)
                 package: web-server-lib
```

```
web-server^ : signature
```

```
(serve) → (-> void)
```

Runs the server and returns a procedure that shuts down the server.

```
(serve-ports ip op) → void
  ip : input-port?
  op : output-port?
```

Serves a single connection represented by the ports *ip* and *op*.

4.1.2 Unit

```
(require web-server/web-server-unit)
                 package: web-server-lib
```

```
web-server@ : (unit/c (web-config^ tcp^)
                      (web-server^))
```

Uses the `web-config^` to construct a `dispatcher/c` function that sets up one virtual host dispatcher, for each virtual host in the `web-config^`, that sequences the following operations:

- Logs the incoming request with the given format to the given file
- Performs HTTP Basic Authentication with the given password file
- Allows the "/conf/refresh-passwords" URL to refresh the password file.
- Allows the "/conf/collect-garbage" URL to call the garbage collector.
- Allows the "/conf/refresh-servlets" URL to refresh the servlets cache.
- Executes servlets mapping URLs to the given servlet root directory under htdocs.
- Serves files under the "/" URL in the given htdocs directory.

Using this dispatcher/c, it loads a dispatching server that provides serve and serve-ports functions that operate as expected.

4.2 Configuration Units

4.2.1 Signature

```
(require web-server/web-config-sig)
                  package: web-server-lib
```

web-config^ : signature

Provides contains the following identifiers.

max-waiting : exact-nonnegative-integer?

Passed to tcp-accept.

virtual-hosts : (string? . -> . host?)

Contains the configuration of individual virtual hosts.

initial-connection-timeout : integer?

Specifies the initial timeout given to a connection.

port : port-number?

Specifies the port to serve HTTP on.

listen-ip : (or/c false/c string?)

Passed to tcp-listen.

make-servlet-namespace : make-servlet-namespace/c

Passed to servlets:make through make-default-path->servlet.

4.2.2 Unit

```
(require web-server/web-config-unit)
```
package: web-server-lib

```
(configuration-table->web-config@
 path
 [#:port port
  #:listen-ip listen-ip
  #:make-servlet-namespace make-servlet-namespace])
→ (unit/c (import) (export web-config^))
 path : path-string?
 port : (or/c false/c port-number?) = #f
 listen-ip : (or/c false/c string?) = #f
 make-servlet-namespace : make-servlet-namespace/c
                        = (make-make-servlet-namespace)
```

Reads the S-expression at *path* and calls `configuration-table-sexpr->web-config@` appropriately.

```
(configuration-table-sexpr->web-config@
 sexpr
 [#:web-server-root web-server-root
  #:port port
  #:listen-ip listen-ip
  #:make-servlet-namespace make-servlet-namespace])
→ (unit/c (import) (export web-config^))
 sexpr : list?
 web-server-root : path-string?
                 = (directory-part default-configuration-table-path)
 port : (or/c false/c port-number?) = #f
 listen-ip : (or/c false/c string?) = #f
 make-servlet-namespace : make-servlet-namespace/c
                        = (make-make-servlet-namespace)
```

Parses *sexpr* as a configuration-table and constructs a `web-config^` unit.

4.3 Configuration Table

```
(require web-server/configuration/configuration-table)
```
package: web-server-lib

This module provides functions for reading, writing, parsing, and printing `configuration-table` structures.

```
default-configuration-table-path : path?
```

The default configuration table S-expression file.

```
configuration-table-sexpr? : (any . -> . boolean?)
```

Equivalent to `list?`.

```
(sexpr->configuration-table sexpr) → configuration-table?
  sexpr : configuration-table-sexpr?
```

This function converts a `configuration-table` from an S-expression.

```
(configuration-table->sexpr ctable)
 → configuration-table-sexpr?
  ctable : configuration-table?
```

This function converts a `configuration-table` to an S-expression.

The configuration table format is:

```
`((port ,integer?)
  (max-waiting ,exact-integer?)
  (initial-connection-timeout ,integer?)
  (default-host-table
    ,host-table-sexpr?)
  (virtual-host-table
   (list ,symbol? ,host-table-sexpr?)
   ...))
```

where a `host-table-sexpr` is:

```
`(host-table
```

```
(default-indices ,string? ...)
(log-format ,symbol?)
(messages
 (servlet-message ,path-string?)
 (authentication-message ,path-string?)
 (servlets-refreshed ,path-string?)
 (passwords-refreshed ,path-string?)
 (file-not-found-message ,path-string?)
 (protocol-message ,path-string?)
 (collect-garbage ,path-string?))
(timeouts
 (default-servlet-timeout ,integer?)
 (password-connection-timeout ,integer?)
 (servlet-connection-timeout ,integer?)
 (file-per-byte-connection-timeout ,integer?)
 (file-base-connection-timeout ,integer))
(paths
 (configuration-root ,path-string?)
 (host-root ,path-string?)
 (log-file-path ,path-string?)
 (file-root ,path-string?)
 (servlet-root ,path-string?)
 (mime-types ,path-string?)
 (password-authentication ,path-string?)))
```

In this syntax, the `'messages` paths are relative to the `'configuration-root` directory. All the paths in `'paths` except for `'servlet-root` are relative to `'host-root` (other than `'host-root` obviously.) The `'servlet-root` path is relative to `'file-root`.

Allowable `'log-format`s are those accepted by `log-format->format`.

Note: You almost always want to leave everything in the `'paths` section the default except the `'host-root`.

```
(read-configuration-table path) → configuration-table?
  path : path-string?
```

This function reads a `configuration-table` from *path*.

```
(write-configuration-table ctable path) → void
  ctable : configuration-table?
  path : path-string?
```

This function writes a `configuration-table` to *path*.

4.4 Configuration Table Structure

```
(require web-server/configuration/configuration-table-structs)
```
package: web-server-lib

This module provides the following structures that represent a standard configuration (see §4.1 “Server Units”) of the Web Server . The contracts on this structure influence the valid types of values in the configuration table S-expression file format described in §4.3 “Configuration Table”.

```
(struct configuration-table (port
                             max-waiting
                             initial-connection-timeout
                             default-host
                             virtual-hosts)
    #:extra-constructor-name make-configuration-table)
  port : port-number?
  max-waiting : exact-nonnegative-integer?
  initial-connection-timeout : natural-number/c
  default-host : host-table?
  virtual-hosts : (listof (cons/c string? host-table?))
```

```
(struct host-table (indices log-format messages timeouts paths)
    #:extra-constructor-name make-host-table)
  indices : (listof string?)
  log-format : symbol?
  messages : messages?
  timeouts : timeouts?
  paths : paths?
```

```
(struct host (indices
              log-format
              log-path
              passwords
              responders
              timeouts
              paths)
    #:extra-constructor-name make-host)
  indices : (listof string?)
  log-format : symbol?
  log-path : (or/c false/c path-string?)
  passwords : (or/c false/c path-string?)
  responders : responders?
```

```
  timeouts : timeouts?
  paths : paths?
```

```
(struct responders (servlet
                    servlet-loading
                    authentication
                    servlets-refreshed
                    passwords-refreshed
                    file-not-found
                    protocol
                    collect-garbage)
    #:extra-constructor-name make-responders)
  servlet : (url? any/c . -> . response?)
  servlet-loading : (url? any/c . -> . response?)
  authentication : (url? (cons/c symbol? string?) . -> . response?)
  servlets-refreshed : (-> response?)
  passwords-refreshed : (-> response?)
  file-not-found : (request? . -> . response?)
  protocol : (url? . -> . response?)
  collect-garbage : (-> response?)
```

```
(struct messages (servlet
                  authentication
                  servlets-refreshed
                  passwords-refreshed
                  file-not-found
                  protocol
                  collect-garbage)
    #:extra-constructor-name make-messages)
  servlet : string?
  authentication : string?
  servlets-refreshed : string?
  passwords-refreshed : string?
  file-not-found : string?
  protocol : string?
  collect-garbage : string?
```

```
(struct timeouts (default-servlet
                  password
                  servlet-connection
                  file-per-byte
                  file-base)
```

```
    #:extra-constructor-name make-timeouts)
  default-servlet : number?
  password : number?
  servlet-connection : number?
  file-per-byte : number?
  file-base : number?
```

```
(struct paths (conf
               host-base
               log
               htdocs
               servlet
               mime-types
               passwords)
    #:extra-constructor-name make-paths)
  conf : (or/c false/c path-string?)
  host-base : (or/c false/c path-string?)
  log : (or/c false/c path-string?)
  htdocs : (or/c false/c path-string?)
  servlet : (or/c false/c path-string?)
  mime-types : (or/c false/c path-string?)
  passwords : (or/c false/c path-string?)
```

4.5 Standard Responders

```
(require web-server/configuration/responders)
```
package: web-server-lib

This module provides some functions that help constructing HTTP responders. These functions are used by the default dispatcher constructor (see §4.1 "Server Units") to turn the paths given in the `configuration-table` into responders for the associated circumstance.

```
(file-response http-code
               short-version
               text-file
               header ...)   → response?
  http-code : natural-number/c
  short-version : string?
  text-file : string?
  header : header?
```

Generates a `response?` with the given *`http-code`* and *`short-version`* as the corresponding fields; with the content of the *`text-file`* as the body; and, with the *`header`*s as, you

guessed it, headers.

This does not cause redirects to a well-known URL, such as "conf/not-found.html", but rather use the contents of "not-found.html" (for example) as its contents. Therefore, any relative URLs in *text-file* are relative to whatever URL file-response is used to respond *to*. Thus, you should probably use absolute URLs in these files.

```
(servlet-loading-responder url exn) → response?
  url : url?
  exn : exn?
```

Gives *exn* to the current-error-handler and response with a stack trace and a "Servlet didn't load" message.

```
(gen-servlet-not-found file) → ((url url?) . -> . response?)
  file : path-string?
```

Returns a function that generates a standard "Servlet not found." error with content from *file*.

```
(servlet-error-responder url exn) → response?
  url : url?
  exn : exn?
```

Gives *exn* to the current-error-handler and response with a stack trace and a "Servlet error" message.

```
(gen-servlet-responder file)
 → ((url url?) (exn any/c) . -> . response?)
  file : path-string?
```

Prints the exn to standard output and responds with a "Servlet error." message with content from *file*.

```
(gen-servlets-refreshed file) → (-> response?)
  file : path-string?
```

Returns a function that generates a standard "Servlet cache refreshed." message with content from *file*.

```
(gen-passwords-refreshed file) → (-> response?)
  file : path-string?
```

Returns a function that generates a standard "Passwords refreshed." message with content from *file*.

```
(gen-authentication-responder file)
 → ((url url?) (header header?) . -> . response?)
  file : path-string?
```

Returns a function that generates an authentication failure error with content from *file* and header as the HTTP header.

```
(gen-protocol-responder file) → ((url url?) . -> . response?)
  file : path-string?
```

Returns a function that generates a "Malformed request" error with content from *file*.

```
(gen-file-not-found-responder file)
 → ((req request?) . -> . response?)
  file : path-string?
```

Returns a function that generates a standard "File not found" error with content from *file*.

```
(gen-collect-garbage-responder file) → (-> response?)
  file : path-string?
```

Returns a function that generates a standard "Garbage collection run" message with content from *file*.

5 Internal APIs

The Web Server is a complicated piece of software and as a result, defines a number of interesting and independently useful sub-components. Some of these are documented here.

5.1 Timers

```
(require web-server/private/timer)
```
package: web-server-lib

This module provides a functionality for running procedures after a given amount of time, that may be extended.

```
(timer-manager? x) → boolean?
  x : any/c
```

Determines if *x* is a timer manager.

```
(struct timer (tm evt expire-seconds action)
    #:extra-constructor-name make-timer)
  tm : timer-manager?
  evt : evt?
  expire-seconds : number?
  action : (-> void)
```

`evt` is an `alarm-evt` that is ready at `expire-seconds`. `action` should be called when this `evt` is ready.

```
(start-timer-manager) → timer-manager?
```

Handles the execution and management of timers.

```
(start-timer tm s action) → timer?
  tm : timer-manager?
  s : number?
  action : (-> void)
```

Registers a timer that runs *action* after *s* seconds.

```
(reset-timer! t s) → void
  t : timer?
  s : number?
```

Changes `t` so that it will fire after `s` seconds.

```
(increment-timer! t s) → void
  t : timer?
  s : number?
```

Changes `t` so that it will fire after `s` seconds from when it does now.

```
(cancel-timer! t) → void
  t : timer?
```

Cancels the firing of `t` ever and frees resources used by `t`.

5.2 Connection Manager

```
(require web-server/private/connection-manager)
```
package: `web-server-lib`

This module provides functionality for managing pairs of input and output ports. We have plans to allow a number of different strategies for doing this.

```
(struct connection (timer i-port o-port custodian close?)
    #:extra-constructor-name make-connection)
  timer : timer?
  i-port : input-port?
  o-port : output-port?
  custodian : custodian?
  close? : boolean?
```

A connection is a pair of ports (`i-port` and `o-port`) that is ready to close after the current job if `close?` is `#t`. Resources associated with the connection should be allocated under `custodian`. The connection will last until `timer` triggers.

```
(connection-manager? x) → boolean?
  x : any/c
```

Determines if *x* is a connection manager.

```
(start-connection-manager) → connection-manager?
```

Runs the connection manager (now just the timer manager).

```
(new-connection cm
                timeout
                i-port
                o-port
                cust
                close?) → connection?
  cm : connection-manager?
  timeout : number?
  i-port : input-port?
  o-port : output-port?
  cust : custodian?
  close? : boolean?
```

Constructs a connection with a timer with a trigger of *timeout* that calls kill-connection!.

```
(kill-connection! c) → void
  c : connection?
```

Closes the ports associated with *c*, kills the timer, and shuts down the custodian.

```
(adjust-connection-timeout! c t) → void
  c : connection?
  t : number?
```

Calls increment-timer! with the timer behind *c* with *t*.

5.3 Serializable Closures

The defunctionalization process of the Web Language (see §3 “Stateless Servlets”) requires an explicit representation of closures that is serializable.

```
(require web-server/lang/serial-lambda)
```
package: web-server-lib

(serial-lambda *formals* *body* ...)

Returns (lambda *formals* *body* ...), except it is serializable.

(serial-case-lambda [*formals* *body* ...] ...)

Returns (case-lambda [*formals* *body* ...] ...), except it is serializable.

5.3.1 Definition Syntax

```
(require web-server/private/define-closure)
```
package: web-server-lib

(define-closure *tag* *formals* (*free-var* ...) *body*)

Defines a closure, constructed with make-tag that accepts a closure that returns freevar ..., that when invoked with *formals* executes *body*.

Here is an example:

```
#lang racket
(require racket/serialize)

(define-closure foo (a b) (x y)
 (+ (- a b)
    (* x y)))

(define f12 (make-foo (lambda () (values 1 2))))
(serialize f12)
'((1) 1 (('page . foo:deserialize-info)) 0 () () (0 1 2))
(f12 6 7)
1
(f12 9 1)
10

(define f45 (make-foo (lambda () (values 4 5))))
(serialize f45)
'((1) 1 (('page . foo:deserialize-info)) 0 () () (0 4 5))
(f45 1 2)
19
(f45 8 8)
20
```

5.4 Cache Table

```
(require web-server/private/cache-table)
```
package: web-server-lib

This module provides a set of caching hash table functions.

```
(make-cache-table) → cache-table?
```

Constructs a cache-table.

```
(cache-table-lookup! ct id mk) → any/c
  ct : cache-table?
  id : symbol?
  mk : (-> any/c)
```

Looks up *id* in *ct*. If it is not present, then *mk* is called to construct the value and add it to *ct*.

```
(cache-table-clear! ct) → void?
  ct : cache-table?
```

Clears all entries in *ct*.

```
(cache-table? v) → boolean?
  v : any/c
```

Determines if *v* is a cache table.

5.5 MIME Types

```
(require web-server/private/mime-types)
```
package: web-server-lib

This module provides function for dealing with `"mime.types"` files.

```
(read-mime-types p) → (hash/c symbol? bytes?)
  p : path-string?
```

Reads the `"mime.types"` file from *p* and constructs a hash table mapping extensions to MIME types.

```
(make-path->mime-type p) → (path? . -> . (or/c false/c bytes?))
  p : path-string?
```

Uses a `read-mime-types` with *p* and constructs a function from paths to their MIME type.

5.6 Serialization Utilities

```
(require web-server/private/mod-map)
                    package: web-server-lib
```

The `racket/serialize` library provides the functionality of serializing values. This module compresses the serialized representation.

```
(compress-serial sv) → list?
  sv : list?
```

Collapses multiple occurrences of the same module in the module map of the serialized representation, *sv*.

```
(decompress-serial csv) → list?
  csv : list?
```

Expands multiple occurrences of the same module in the module map of the compressed serialized representation, *csv*.

5.7 URL Param

```
(require web-server/private/url-param)
                    package: web-server-lib
```

The Web Server needs to encode information in URLs. If this data is stored in the query string, than it will be overridden by browsers that make GET requests to those URLs with more query data. So, it must be encoded in URL params. This module provides functions for helping with this process.

```
(insert-param u k v) → url?
  u : url?
  k : string?
  v : string?
```

Associates *k* with *v* in the final URL param of *u*, overwritting any current binding for *k*.

```
(extract-param u k) → (or/c string? false/c)
  u : url?
  k : string?
```

Extracts the string associated with *k* in the final URL param of *u*, if there is one, returning `#f` otherwise.

5.8 GZip

```
(require web-server/private/gzip)
```
package: web-server-lib

The Web Server provides a thin wrapper around `file/gzip` and `file/gunzip`.

```
(gzip/bytes ib) → bytes?
  ib : bytes?
```

GZips *ib* and returns the result.

```
(gunzip/bytes ib) → bytes?
  ib : bytes?
```

GUnzips *ib* and returns the result.

5.9 Miscellaneous Utilities

```
(require web-server/private/util)
```
package: web-server-lib

```
(bytes-ci=? b1 b2) → boolean?
  b1 : bytes?
  b2 : bytes?
```

Compares two bytes case insensitively.

```
(url-replace-path proc u) → url?
  proc : ((listof path/param?) . -> . (listof path/param?))
  u : url?
```

Replaces the URL path of *u* with *proc* of the former path.

```
(url-path->string url-path) → string?
  url-path : (listof path/param?)
```

Formats *url-path* as a string with "/" as a delimiter and no params.

```
(explode-path* p) → (listof path-piece?)
  p : path-string?
```

Like normalize-path, but does not resolve symlinks.

```
(path-without-base base p) → (listof path-piece?)
  base : path-string?
  p : path-string?
```

Returns, as a list, the portion of *p* after *base*, assuming *base* is a prefix of *p*.

```
(directory-part p) → path?
  p : path-string?
```

Returns the directory part of *p*, returning (current-directory) if it is relative.

```
(build-path-unless-absolute base p) → path?
  base : path-string?
  p : path-string?
```

Prepends *base* to *p*, unless *p* is absolute.

```
(network-error s fmt v ...) → void
  s : symbol?
  fmt : string?
  v : any/c
```

Like error, but throws a exn:fail:network.

```
(exn->string exn) → string?
  exn : (or/c exn? any/c)
```

Formats *exn* with (error-display-handler) as a string.

6 Troubleshooting and Tips

6.1 How do I use Apache with the Racket Web Server?

You may want to put Apache in front of your Racket Web Server application. Apache can rewrite and proxy requests for a private (or public) Racket Web Server:

```
RewriteEngine on
RewriteRule ^(.*)$ http://localhost:8080/$1 [P,NE]
```

The first argument to `RewriteRule` is a match pattern. The second is how to rewrite the URL. The bracketed part contains flags that specify the type of rewrite, in this case the `P` flag instructs Apache to proxy the request. (If you do not include this, Apache will return an HTTP Redirect response and the client will make a second request to `localhost:8080` which will not work on a different machine.) In addition, the `NE` flag is needed to avoid escaping parts of the URL — without it, a `;` is escaped as `%3B` which will break the proxied request.

See Apache's documentation for more details on RewriteRule.

6.2 Can the server create a PID file?

The server has no option for this, but you can add it very easily. There's two techniques.

First, if you use a UNIX platform, in your shell startup script you can use

```
echo $$ > PID
exec run-web-server
```

Using `exec` will reuse the same process, and therefore, the PID file will be accurate.

Second, if you want to make your own Racket start-up script, you can write:

```
(require mzlib/os)
(with-output-to-file your-pid-file (lambda () (write (getpid))))
(start-server)
```

6.3 How do I set up the server to use HTTPS?

This requires an SSL certificate and private key. This is very platform specific, but we will provide the details for using OpenSSL on UNIX:

```
openssl genrsa -des3 -out private-key.pem 1024
```

This will generate a new private key, but it will have a passphrase on it. You can remove this via:

```
  openssl rsa -in private-key.pem -out private-key.pem

  chmod 400 private-key.pem
```

Now, we generate a self-signed certificate:

```
  openssl req -new -x509 -nodes -sha1 -days 365 -key private-key.pem
> server-cert.pem
```

(Each certificate authority has different instructions for generating certificate signing requests.)

We can now start the server with:

```
  plt-web-server --ssl
```

The Web Server will start on port 443 (which can be overridden with the `-p` option) using the `"private-key.pem"` and `"server-cert.pem"` we've created.

6.4 How do I limit the number of requests serviced at once by the Web Server?

Refer to §2.14 "Limiting Requests".

Index