Web Server: PLT HTTP Server

Version 4.1.4

Jay McCarthy <jay at plt-scheme dot org>

January 20, 2009

The Web Server collection provides libraries that can be used to develop Web applications in Scheme.

Contents

1 Running the Web Server

There are a number of ways to run the Web Server. They are given in order of simplest to most advanced.

1.1 Instant Servlets

```
#lang web-server/insta
```

The fastest way to get a servlet running in the Web server is to use the ”Insta” language in DrScheme. Enter the following into DrScheme:

```
#lang web-server/insta

(define (start request)
  `(html (head (title "Hello world!"))
         (body (p "Hey out there!"))))
```

And press Run. A Web browser will open up showing your new servlet.

Behind the scenes, DrScheme has used `serve/servlet` to start a new server that uses your `start` function as the servlet. You are given the entire `web-server/servlet` API.

1.1.1 Customization API

```
(require web-server/insta/insta)
```

The following API is provided to customize the server instance:

`(no-web-browser)` → `void`

Calling this will instruct DrScheme to *not* start a Web browser when you press Run.

`(static-files-path` *`path`*`)` → `void`
  `path` : `path-string?`

This instructs the Web server to serve static files, such as stylesheet and images, from *`path`*.

1.2 Simple Single Servlet Servers

```
(require web-server/servlet-env)
```

The Web Server provides a way to quickly configure and start a server instance.

Here's a simple example:

```
#lang scheme
(require web-server/servlet
         web-server/servlet-env)

(define (my-app request)
  `(html (head (title "Hello world!"))
         (body (p "Hey out there!"))))

(serve/servlet my-app)
```

Suppose you'd like to change the port to something else, change the last line to:

```
(serve/servlet my-app
               #:port 8080)
```

By default the URL for your servlet is `"http://localhost:8000/servlets/standalone.ss"`, suppose you wanted it to be `"http://localhost:8000/hello.ss"`:

```
(serve/servlet my-app
               #:servlet-path "/hello.ss")
```

Suppose you wanted it to capture top-level requests:

```
(serve/servlet my-app
               #:servlet-path "/")
```

Or, perhaps just some nice top-level name:

```
(serve/servlet my-app
               #:servlet-path "/main")
```

Suppose you wanted to use a style-sheet (`"style.css"`) found on your Desktop (`"/Users/jay/Desktop/"`):

```
(serve/servlet my-app
               #:extra-files-paths
               (list
                (build-path "/Users/jay/Desktop")))
```

These files are served *in addition* to those from the `#:server-root-path` `"htdocs"` directory. Notice that you may pass any number of extra paths.

Suppose you would like to start a server for a stateless Web servlet `"servlet.ss"` that provides `start`:

```
#lang scheme
(require "servlet.ss"
         web-server/servlet-env)

(serve/servlet start #:stateless? #t)
```

Note: If you put the call to `serve/servlet` in the module like normal, strange things will happen because of the way the top-level interacts with continuations. (Read: Don't do it.)

If you want to use `serve/servlet` in a start up script for a Web server, and don't want a browser opened or the DrScheme banner printed, then you can write:

```
(serve/servlet my-app
               #:command-line? #t)
```

```
(serve/servlet
 start
 [#:command-line? command-line?
  #:launch-browser? launch-browser?
  #:quit? quit?
  #:banner? banner?
  #:listen-ip listen-ip
  #:port port
  #:ssl? ssl?
  #:servlet-path servlet-path
  #:servlet-regexp servlet-regexp
  #:stateless? stateless?
  #:manager manager
  #:servlet-namespace servlet-namespace
  #:server-root-path server-root-path
  #:extra-files-paths extra-files-paths
  #:servlets-root servlets-root
  #:servlet-current-directory servlet-current-directory
  #:file-not-found-responder file-not-found-responder
  #:mime-types-path mime-types-path
  #:log-file log-file
  #:log-format log-format])
→ void
 start : (request? . -> . response?)
 command-line? : boolean? = #f
 launch-browser? : boolean? = (not command-line?)
 quit? : boolean? = (not command-line?)
 banner? : boolean? = (not command-line?)
```

```
listen-ip : (or/c false/c string?) = "127.0.0.1"
port : number? = 8000
ssl? : boolean? = #f
servlet-path : string? = "/servlets/standalone.ss"
servlet-regexp : regexp? = (regexp
                            (format
                             "^~a$"
                             (regexp-quote servlet-path)))
stateless? : boolean? = #f
manager : manager?
        = (make-threshold-LRU-manager #f (* 1024 1024 64))
servlet-namespace : (listof module-path?) = empty
server-root-path : path-string? = default-server-root-path
extra-files-paths : (listof path-string?)
                  = (list (build-path server-root-path "htdocs"))
servlets-root : path-string?
              = (build-path server-root-path "htdocs")
servlet-current-directory : path-string? = servlets-root
file-not-found-responder : (request? . -> . response?)
                         = (gen-file-not-found-responder
                            (build-path
                             server-root-path
                             "conf"
                             "not-found.html"))
mime-types-path : path-string? = ....
log-file : (or/c false/c path-string?) = #f
log-format : symbol? = 'apache-default
```

This sets up and starts a fairly default server instance.

start is loaded as a servlet and responds to requests that match *servlet-regexp*. The current directory of servlet execution is *servlets-root*.

If *launch-browser?* is true, then a web browser is opened to "http://localhost:<port><servlet-path>".

If *quit?* is true, then the URL "/quit" ends the server.

If *stateless?* is true, then the servlet is run as a stateless web-server module.

Advanced users may need the following options:

The server listens on *listen-ip* and port *port*.

If *ssl?* is true, then the server runs in HTTPS mode with "<server-root-path>/server-cert.pem" and "<server-root-path>/private-key.pem" as the certificates and private keys

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 modules specified by *servlet-namespace* are shared with other servlets.

The server files are rooted at *server-root-path* (which is defaultly the distribution root.) File paths, in addition to the "htdocs" directory under *server-root-path* may be provided with *extra-files-paths*. These paths are checked first, in the order they appear in the list.

Other servlets are served from *servlets-root*.

If a file cannot be found, *file-not-found-responder* is used to generate an error response.

If *banner?* is true, then an informative banner is printed. You may want to use this when running from the command line, in which case the *command-line?* option controls similar options.

MIME types are looked up at *mime-types-path*. By default the "mime.types" file in the *server-root-path* is used, but if that file does not exist, then the file that ships with the Web Server is used instead. Of course, if a path is given, then it overrides this behavior.

If *log-file* is given, then it used to log requests using *log-format* as the format. Allowable formats are those allowed by log-format->format.

1.3 Command-line Tools

One command-line utility is provided with the Web Server:

```
plt-web-server [-f <file-name> -p <port> -a <ip-address> --ssl]
```

The optional file-name argument specifies the path to a configuration-table S-expression (see §3.1.2 "Configuration Table".) If this is not provided, the default configuration shipped with the server is used. The optional port and ip-address arguments override the corresponding portions of the configuration-table. If the SSL option is provided, then the server uses HTTPS with "server-cert.pem" and "private-key.pem" in the current directory, with 443 as the default port. (See the openssl module for details on the SSL implementation.)

The configuration-table is given to configuration-table->web-config@ and used to construct a web-config^ unit, and is linked with the web-server@ unit. The resulting unit is invoked, and the server runs until the process is killed.

To run the web server with MrEd, use

```
mred -l- web-server/gui [-f <file-name> -p <port> -a <ip-address>]
```

1.4 Functional

```
(require web-server/web-server)
```

"web-server.ss" provides a number of functions for easing embedding of the Web Server in other applications, or loading a custom dispatcher.

```
(serve
 #:dispatch dispatch
 [#:tcp@ tcp@
  #:port port
  #:listen-ip listen-ip
  #:max-waiting max-waiting
  #:initial-connection-timeout initial-connection-timeout])
→ (-> void)
 dispatch : dispatcher/c
 tcp@ : tcp-unit^ = raw:tcp@
 port : integer? = 80
 listen-ip : (or/c string? false/c) = #f
 max-waiting : integer? = 40
 initial-connection-timeout : integer? = 60
```

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

The #:tcp@ keyword is provided for building an SSL server. See §4.6 "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
 [#:tcp@ tcp@
  #:ports ports
  #:listen-ip listen-ip
  #:max-waiting max-waiting
  #:initial-connection-timeout initial-connection-timeout])
→ (-> void)
 dispatch : dispatcher/c
 tcp@ : tcp-unit^ = raw:tcp@
 ports : (listof integer?) = (list 80)
 listen-ip : (or/c string? false/c) = #f
 max-waiting : integer? = 40
 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
 [#:tcp@ tcp@
  #:ips+ports ips+ports
  #:max-waiting max-waiting
  #:initial-connection-timeout initial-connection-timeout])
→ (-> void)
 dispatch : dispatcher/c
 tcp@ : tcp-unit^ = raw:tcp@
 ips+ports : (listof (cons/c (or/c string? false/c) (listof integer?)))
           = (list (cons #f (list 80)))
 max-waiting : integer? = 40
 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@ : web-config^
  tcp@ : tcp-unit^ = 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.

2 Writing Servlets

2.1 Defining a Servlet

A *servlet* is a module with particular exports. There three kinds of servlets.

2.1.1 Version 1 Servlets

```
interface-version : (one-of/c 'v1)
```

This indicates that the servlet is a version one servlet.

```
timeout : integer?
```

This number is used as the `continuation-timeout` argument to a timeout-based continuation manager used for this servlet. (See §2.12.3 "Timeouts".) (i.e., you do not have a choice of the manager for this servlet and will be given a timeout-based manager.)

```
(start initial-request) → response?
  initial-request : request?
```

This function is called when an instance of this servlet is started. The argument is the HTTP request that initiated the instance.

An example version 1 module:

```
#lang scheme

(define interface-version 'v1)
(define timeout (* 60 60 24))
(define (start req)
  `(html (head (title "Hello World!"))
         (body (h1 "Hi Mom!"))))
```

These servlets should use the `web-server/servlet` API.

2.1.2 Version 2 Servlets

```
interface-version : (one-of/c 'v2)
```

This indicates that the servlet is a version two servlet.

`manager : manager?`

The manager for the continuations of this servlet. See §2.12 “Continuation Managers” for options.

`(start initial-request) → response?`
`  initial-request : request?`

This function is called when an instance of this servlet is started. The argument is the HTTP request that initiated the instance.

An example version 2 module:

```
#lang scheme
(require web-server/managers/none)

(define interface-version 'v2)
(define manager
  (create-none-manager
   (lambda (req)
     `(html (head (title "No Continuations Here!"))
            (body (h1 "No Continuations Here!"))))))
(define (start req)
  `(html (head (title "Hello World!"))
         (body (h1 "Hi Mom!"))))
```

These servlets should use the `web-server/servlet` API.

2.1.3 Stateless Servlets

`interface-version : (one-of/c 'stateless)`

This indicates that the servlet is a stateless servlet.

`(start initial-request) → response?`
`  initial-request : request?`

This function is called when an instance of this servlet is started. The argument is the HTTP request that initiated the instance.

An example 'stateless servlet module:

```
#lang web-server
(define interface-version 'stateless)
(define (start req)
  `(html (body (h2 "Look ma, no state!"))))
```

The web-server language automatically provides the web-server/lang/lang-api API.

Usage Considerations

```
#lang web-server
```

A servlet has the following process performed on it automatically:

- All uses of letrec are removed and replaced with equivalent uses of let and imperative features. ("lang/elim-letrec.ss")
- The program is converted into ANF (Administrative Normal Form), making all continuations explicit. ("lang/anormal.ss")
- All continuations (and other continuations marks) are recorded in the continuation marks of the expression they are the continuation of. ("lang/elim-callcc.ss")
- All calls to external modules are identified and marked. ("lang/elim-callcc.ss")
- All uses of call/cc are removed and replaced with equivalent gathering of the continuations through the continuation-marks. ("lang/elim-callcc.ss")
- The program is defunctionalized with a serializable data-structure for each anonymous lambda. ("lang/defun.ss")

This process allows the continuations captured by your servlet to be serialized. This means they may be stored on the client's browser or the server's disk. Thus, your servlet has no cost to the server other than execution. This is very attractive if you've used Scheme servlets and had memory problems.

This process IS defined on all of PLT Scheme and occurs AFTER macro-expansion, so you are free to use all interesting features of PLT Scheme. However, there are some considerations you must make.

First, this process drastically changes the structure of your program. It will create an immense number of lambdas and structures your program did not normally contain. The performance implication of this has not been studied with PLT Scheme. However, it is theoretically a benefit. The main implications would be due to optimizations MzScheme attempts to perform that will no longer apply. Ideally, your program should be optimized first.

Second, the defunctionalization process is sensitive to the syntactic structure of your program. Therefore, if you change your program in a trivial way, for example, changing a constant, then all serialized continuations will be obsolete and will error when deserialization is attempted. This is a feature, not a bug!

Third, the values in the lexical scope of your continuations must be serializable for the continuations itself to be serializable. This means that you must use `define-serializable-struct` rather than `define-struct`, and take care to use modules that do the same. Similarly, you may not use `parameterize`, because parameterizations are not serializable.

Fourth, and related, this process only runs on your code, not on the code you `require`. Thus, your continuations—to be capturable—must not be in the context of another module. For example, the following will not work:

```
(define requests
  (map (lambda (rg) (send/suspend/url rg))
       response-generators))
```

because `map` is not transformed by the process. However, if you defined your own `map` function, there would be no problem.

Fifth, the store is NOT serialized. If you rely on the store you will be taking huge risks. You will be assuming that the serialized continuation is invoked before the server is restarted or the memory is garbage collected.

This process is derived from the paper Continuations from Generalized Stack Inspection. We thank Greg Pettyjohn for his initial implementation of this algorithm.

2.2 APIs

There are two API sets provided by the Web Server. One is for standard servlets, the other is for stateless servlets.

2.2.1 Standard API

```
(require web-server/servlet)
```

This API provides:

- `web-server/servlet/web-cells`,
- `web-server/http/bindings`,
- `web-server/http`,

- web-server/servlet/servlet-structs, and
- web-server/servlet/web.

2.2.2 Stateless API

```
(require web-server/lang/lang-api)
```

This API provides:

- net/url,
- web-server/http,
- web-server/lang/abort-resume,
- web-server/lang/web,
- web-server/lang/web-cells,
- web-server/lang/web-param, and
- web-server/lang/file-box.

2.3 Common Contracts

```
(require web-server/servlet/servlet-structs)
```

"servlet/servlet-structs.ss" provides a number of contracts for use in servlets.

---

k-url? : contract?

Equivalent to string?.

Example: "http://localhost:8080/servlets;1*1*20131636/examples/add.ss"

---

response-generator/c : contract?

Equivalent to (-> k-url? response?).

Example:

```
(lambda (k-url)
  `(html
```

```
      (body
       (a ([href ,k-url])
          "Click Me to Invoke the Continuation!"))))
```

`expiration-handler/c` : `contract?`

Equivalent to `(or/c false/c (-> request? response?))`.

Example:

```
(lambda (req)
  `(html (head (title "Expired"))
         (body (h1 "Expired")
               (p "This URL has expired. "
                  "Please return to the home page."))))
```

`embed/url/c` : `contract?`

Equivalent to `(opt-> ((-> request? any/c)) (expiration-handler/c) string?)`.

This is what `send/suspend/dispatch` gives to its function argument.

2.4 HTTP

```
(require web-server/http)
```

The Web Server implements many HTTP RFCs that are provided by this module.

2.4.1 Requests

```
(require web-server/http/request-structs)
```

```
(struct header (field value))
  field : bytes?
  value : bytes?
```

Represents a header of `field` to `value`.

`(headers-assq` *id heads*`)` → `(or/c false/c header?)`

```
  id : bytes?
  heads : (listof header?)
```

Returns the header with a field equal to *id* from *heads* or #f.

```
(headers-assq* id heads) → (or/c false/c header?)
  id : bytes?
  heads : (listof header?)
```

Returns the header with a field case-insensitively equal to *id* from *heads* or #f.

You almost **always** want to use this, rather than headers-assq because Web browsers may send headers with arbitrary casing.

```
(struct binding (id))
  id : bytes?
```

Represents a binding of id.

```
(struct (binding:form binding) (value))
  value : bytes?
```

Represents a form binding of id to value.

```
(struct (binding:file binding) (filename headers content))
  filename : bytes?
  headers : (listof header?)
  content : bytes?
```

Represents the uploading of the file filename with the id id and the content content, where headers are the additional headers from the MIME envelope the file was in. (For example, the #"Content-Type" header may be included by some browsers.)

```
(bindings-assq id binds) → (or/c false/c binding?)
  id : bytes?
  binds : (listof binding?)
```

Returns the binding with an id equal to *id* from *binds* or #f.

```
(struct request (method
                 uri
                 headers/raw
                 bindings/raw
                 post-data/raw
                 host-ip
                 host-port
                 client-ip))
  method : symbol?
  uri : url?
  headers/raw : (listof header?)
  bindings/raw : (listof binding?)
  post-data/raw : (or/c false/c bytes?)
  host-ip : string?
  host-port : number?
  client-ip : string?
```

An HTTP `method` request to `uri` from `client-ip` to the server at `host-ip`:`host-port` with `headers/raw` headers, `bindings/raw` GET and POST queries and `post-data/raw` POST data.

You are **unlikely to need to construct** a request struct.

Here is an example typical of what you will find in many applications:

```
(define (get-number req)
  (match
    (bindings-assq
      #"number"
      (request-bindings/raw req))
    [(? binding:form? b)
     (string->number
      (bytes->string/utf-8
       (binding:form-value b)))]
    [_
     (get-number (request-number))]))
```

2.4.2 Bindings

```
(require web-server/http/bindings)
```

These functions, while convenient, could introduce subtle bugs into your application. Examples: that they are case-insensitive could introduce a bug; if the data submitted is not in UTF-8 format, then the conversion to a string will fail; if an attacker submits a form field as if it were a file, when it is not, then the `request-bindings` will hold a `bytes?` object

and your program will error; and, for file uploads you lose the filename. **Therefore, we recommend against their use, but they are provided for compatibility with old code.**

```
(request-bindings req)
 → (listof (or/c (cons/c symbol? string?)
                 (cons/c symbol? bytes?)))
  req : request?
```

Translates the `request-bindings/raw` of *req* by interpreting `bytes?` as `string?`s, except in the case of `binding:file` bindings, which are left as is. Ids are then translated into lowercase symbols.

```
(request-headers req) → (listof (cons/c symbol? string?))
  req : request?
```

Translates the `request-headers/raw` of *req* by interpreting `bytes?` as `string?`s. Ids are then translated into lowercase symbols.

```
(extract-binding/single id binds) → string?
  id : symbol?
  binds : (listof (cons/c symbol? string?))
```

Returns the single binding associated with *id* in the a-list *binds* if there is exactly one binding. Otherwise raises `exn:fail`.

```
(extract-bindings id binds) → (listof string?)
  id : symbol?
  binds : (listof (cons/c symbol? string?))
```

Returns a list of all the bindings of *id* in the a-list *binds*.

```
(exists-binding? id binds) → boolean?
  id : symbol?
  binds : (listof (cons/c symbol? string))
```

Returns `#t` if *binds* contains a binding for *id*. Otherwise, `#f`.

Here is an example typical of what you will find in many applications:

```
(define (get-number req)
  (string->number
   (extract-binding/single
```

```
      'number
      (request-bindings req))))
```

2.4.3 Responses

```
(require web-server/http/response-structs)
```

```
(struct response/basic (code message seconds mime headers))
  code : number?
  message : string?
  seconds : number?
  mime : bytes?
  headers : (listof header?)
```

A basic HTTP response containing no body. `code` is the response code, `message` the message, `seconds` the generation time, `mime` the MIME type of the file, and `extras` are the extra headers, in addition to those produced by the server.

Example:

```
  (make-response/basic
   301 "Moved Permanently"
   (current-seconds) TEXT/HTML-MIME-TYPE
   (list (make-header #"Location"
                      #"http://www.plt-scheme.org/downloads")))
```

```
(struct (response/full response/basic) (body))
  body : (listof (or/c string? bytes?))
```

As with `response/basic`, except with `body` as the response body.

Example:

```
  (make-response/full
   301 "Moved Permanently"
   (current-seconds) TEXT/HTML-MIME-TYPE
   (list (make-header #"Location"
                      #"http://www.plt-scheme.org/downloads"))
   (list #"<html><body><p>"
         #"Please go to <a href=\""
         #"http://www.plt-scheme.org/downloads"
         #"\">here</a> instead."
         #"</p></body></html>"))
```

```
(struct (response/incremental response/basic) (generator))
  generator : ((() (listof (or/c bytes? string?)) . ->* . any) . -> . any)
```

As with response/basic, except with generator as a function that is called to generate the response body, by being given an output-response function that outputs the content it is called with.

Here is a short example:

```
(make-response/incremental
  200 "OK" (current-seconds)
  #"application/octet-stream"
  (list (make-header #"Content-Disposition"
                     #"attachement; filename=\"file\""))
  (lambda (send/bytes)
    (send/bytes #"Some content")
    (send/bytes)
    (send/bytes #"Even" #"more" #"content!")
    (send/bytes "Now we're done")))
```

```
(response? v) → boolean?
  v : any/c
```

Checks if v is a valid response. A response is either:

- A response/basic structure.
- A value matching the contract (cons/c (or/c bytes? string?) (listof (or/c bytes? string?))).
- A value matching xexpr?.

```
TEXT/HTML-MIME-TYPE : bytes?
```

Equivalent to #"text/html; charset=utf-8".

Warning: If you include a Content-Length header in a response that is inaccurate, there **will be an error** in transmission that the server **will not catch**.

2.4.4 Redirect

```
(require web-server/http/redirect)
```

```
(redirect-to  uri
              [perm/temp
               #:headers headers]) → response?
  uri : string?
  perm/temp : redirection-status? = temporarily
  headers : (listof header?) = (list)
```

Generates an HTTP response that redirects the browser to *uri*, while including the *headers* in the response.

Example: `(redirect-to "http://www.add-three-numbers.com" permanently)`

```
(redirection-status? v) → boolean?
  v : any/c
```

Determines if *v* is one of the following values.

```
permanently : redirection-status?
```

A `redirection-status?` for permanent redirections.

```
temporarily : redirection-status?
```

A `redirection-status?` for temporary redirections.

```
see-other : redirection-status?
```

A `redirection-status?` for ”see-other” redirections.

2.4.5 Basic Authentication

```
(require web-server/http/basic-auth)
```

An implementation of HTTP Basic Authentication.

```
(extract-user-pass heads)
 → (or/c false/c (cons/c bytes? bytes?))
  heads : (listof header?)
```

Returns a pair of the username and password from the authentication header in *heads* if

they are present, or `#f`.

Example: `(extract-user-pass (request-headers/raw req))` might return `(cons #"aladin" #"open sesame")`.

2.5 Web Interaction

```
(require web-server/servlet/web)
```

The `web-server/servlet/web` library provides the primary functions of interest for the servlet developer.

`(send/back` *response*`)` → `void?`
  response : `response?`

Sends *response* to the client. No continuation is captured, so the servlet is done.

Example:

```
(send/back
 `(html
   (body
    (h1 "The sum is: "
        ,(+ first-number
            second-number)))))
```

`(send/suspend` *make-response* [*exp*]`)` → `request?`
  make-response : `response-generator/c`
  exp : `expiration-handler/c`
      = `(current-servlet-continuation-expiration-handler)`

Captures the current continuation, stores it with *exp* as the expiration handler, and binds it to a URL. *make-response* is called with this URL and is expected to generate a `response?`, which is sent to the client. If the continuation URL is invoked, the captured continuation is invoked and the request is returned from this call to `send/suspend`.

Example:

```
(send/suspend
 (lambda (k-url)
   `(html (head (title "Enter a number"))
          (body
           (form ([action ,k-url])
                 "Enter a number: "
```

```
                (input ([name "number"]))
                (input ([type "submit"])))))))
```

When this form is submitted by the browser, the request will be sent to the URL generated by send/suspend. Thus, the request will be "returned" from send/suspend to the continuation of this call.

(send/suspend/dispatch *make-response*) → any/c
  make-response : (embed/url/c . -> . response?)

Calls *make-response* with a function (embed/url) that, when called with a procedure from request? to any/c will generate a URL, that when invoked will call the function with the request? object and return the result to the caller of send/suspend/dispatch. Therefore, if you pass embed/url the identity function, send/suspend/dispatch devolves into send/suspend:

```
(define (send/suspend response-generator)
  (send/suspend/dispatch
   (lambda (embed/url)
     (response-generator (embed/url (lambda (x) x))))))
```

Use send/suspend/dispatch when there are multiple 'logical' continuations of a page. For example, we could either add to a number or subtract from it:

```
(define (count-dot-com i)
  (count-dot-com
   (send/suspend/dispatch
    (lambda (embed/url)
      `(html
        (head (title "Count!"))
        (body
         (h2 (a ([href
                  ,(embed/url
                    (lambda (req)
                      (sub1 i)))])
                "-"))
         (h1 ,(number->string i))
         (h2 (a ([href
                  ,(embed/url
                    (lambda (req)
                      (add1 i)))])
                "+")))))))))
```

Notice that in this example the result of the handlers are returned to the continuation of send/suspend/dispatch. However, it is very common that the return value of send/suspend/dispatch is irrevelant in your application and you may think of it as "em-

bedding" value-less callbacks. Here is the same example in this style:

```
(define (count-dot-com i)
  (send/suspend/dispatch
   (lambda (embed/url)
     `(html
       (head (title "Count!"))
       (body
        (h2 (a ([href
                 ,(embed/url
                   (lambda (req)
                     (count-dot-com (sub1 i))))])
               "-"))
        (h1 ,(number->string i))
        (h2 (a ([href
                 ,(embed/url
                   (lambda (req)
                     (count-dot-com (add1 i))))])
               "+")))))))
```

```
(send/forward make-response [exp]) → request?
  make-response : response-generator/c
  exp : expiration-handler/c
      = (current-servlet-continuation-expiration-handler)
```

Calls `clear-continuation-table!`, then `send/suspend`.

Use this if the user can logically go 'forward' in your application, but cannot go backward.

```
(send/finish response) → void?
  response : response?
```

Calls `clear-continuation-table!`, then `send/back`.

Use this if the user is truly 'done' with your application. For example, it may be used to display the post-logout page:

```
(send/finish
 `(html (head (title "Logged out"))
        (body (p "Thank you for using the services "
                 "of the Add Two Numbers, Inc."))))
```

```
(redirect/get) → request?
```

Calls send/suspend with redirect-to.

This implements the Post-Redirect-Get pattern. Use this to prevent the Refresh button from duplicating effects, such as adding items to a database.

(redirect/get/forget) → request?

Calls send/forward with redirect-to.

---

current-servlet-continuation-expiration-handler : (parameter/c expiration-handler/c)

Holds the expiration-handler/c to be used when a continuation captured in this context is expired, then looked up.

Example:

```
(parameterize
    ([current-servlet-continuation-expiration-handler
      (lambda (req)
        `(html (head (title "Custom Expiration!"))))])
  (send/suspend
   ....))
```

(clear-continuation-table!) → void?

Calls the servlet's manager's clear-continuation-table! function. Normally, this deletes all the previously captured continuations.

```
(with-errors-to-browser send/finish-or-back
                        thunk)              → any
  send/finish-or-back : (response? . -> . request?)
  thunk : (-> any)
```

Calls *thunk* with an exception handler that generates an HTML error page and calls *send/finish-or-back*.

Example:

```
(with-errors-to-browser
 send/back
 (lambda ()
   (/ 1 (get-number (request-number)))))
```

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

Calls the servlet's manager's `adjust-timeout!` function.

Warning: This is deprecated and will be removed in a future release.

```
(continuation-url? u)
 → (or/c false/c (list/c number? number? number?))
  u : url?
```

Checks if *u* is a URL that refers to a continuation, if so returns the instance id, continuation id, and nonce.

2.6 Stateless Web Interaction

2.6.1 Low Level

```
(require web-server/lang/abort-resume)
```

```
(send/suspend response-generator) → any
  response-generator : (continuation? . -> . any)
```

Captures the current continuation in a serializable way and calls *response-generator* with it, returning the result.

2.6.2 High Level

```
(require web-server/lang/web)
```

```
(send/suspend/url response-generator) → request?
  response-generator : (url? . -> . response?)
```

Captures the current continuation. Serializes it and stuffs it into a URL. Calls *response-generator* with this URL and delivers the response to the client. If the URL is invoked the request is returned to this continuation.

```
(send/suspend/hidden response-generator) → request?
  response-generator : (url? xexpr? . -> . response?)
```

Captures the current continuation. Serializes it and generates an INPUT form that includes the serialization as a hidden form. Calls *response-generator* with this URL and form field and delivers the response to the client. If the URL is invoked with form data containing the hidden form, the request is returned to this continuation.

Note: The continuation is NOT stuffed.

```
(send/suspend/dispatch make-response) → any/c
  make-response : (embed/url/c . -> . response?)
```

Calls *make-response* with a function that, when called with a procedure from `request?` to `any/c` will generate a URL, that when invoked will call the function with the `request?` object and return the result to the caller of `send/suspend/dispatch`.

```
(redirect/get) → request?
```

See `web-server/servlet/web`.

2.6.3 Stuff URL

```
(require web-server/lang/stuff-url)
```

`"lang/stuff-url.ss"` provides an interface for "stuffing" serializable values into URLs. Currently there is a particular hard-coded behavior, but we hope to make it more flexible in the future.

```
(stuff-url v u) → url?
  v : serializable?
  u : url?
```

Returns a URL based on *u* with *v* serialized and "stuffed" into it. The following steps are applied until the URL is short enough to be accepted by IE.

- Put the plain-text serialization in the URL.
- Compress the serialization with `file/gzip` into the URL.
- Compute the MD5 of the compressed seralization and write it to `"$HOME/.urls/M"` where 'M' is the MD5. 'M' is then placed in the URL

```
(stuffed-url? u) → boolean?
  u : url?
```

Checks if *u* appears to be produced by stuff-url.

```
(unstuff-url u) → serializable?
  u : url?
```

Extracts the value previously serialized into *u* by stuff-url.

In the future, we will offer the facilities to:

- Optionally use the content-addressed storage.
- Use different hashing algorithms for the CAS.
- Encrypt the serialized value.
- Only use the CAS if the URL would be too long. (URLs may only be 1024 characters.)

2.7 Web Cells

```
(require web-server/servlet/web-cells)
```

The web-server/servlet/web-cells library provides the interface to Web cells.

A Web cell is a kind of state defined relative to the *frame tree*. The frame-tree is a mirror of the user's browsing session. Every time a continuation is invoked, a new frame (called the *current frame*) is created as a child of the current frame when the continuation was captured.

You should use Web cells if you want an effect to be encapsulated in all interactions linked from (in a transitive sense) the HTTP response being generated. For more information on their semantics, consult the paper "Interaction-Safe State for the Web".

```
(web-cell? v) → boolean?
  v : any/c
```

Determines if *v* is a web-cell.

```
(make-web-cell v) → web-cell?
  v : any/c
```

Creates a web-cell with a default value of *v*.

```
(web-cell-ref wc) → any/c
```

```
  wc : web-cell?
```

Looks up the value of `wc` found in the nearest frame.

```
(web-cell-shadow wc v) → void
  wc : web-cell?
  v : any/c
```

Binds `wc` to `v` in the current frame, shadowing any other bindings to `wc` in the current frame.

Below is an extended example that demonstrates how Web cells allow the creation of reusable Web abstractions without requiring global transformations of the program into continuation or store passing style.

```
#lang web-server/insta

(define (start initial-request)
 (define counter1 (make-counter))
 (define counter2 (make-counter))
 (define include1 (include-counter counter1))
 (define include2 (include-counter counter2))
 (send/suspend/dispatch
  (lambda (embed/url)
    `(html
      (body (h2 "Double Counters")
            (div (h3 "First")
                 ,(include1 embed/url))
            (div (h3 "Second")
                 ,(include2 embed/url)))))))

(define (make-counter)
 (make-web-cell 0))

(define (include-counter a-counter)
 (let/cc k
   (let loop ()
     (k
      (lambda (embed/url)
        `(div (h3 ,(number->string (web-cell-ref a-counter)))
              (a ([href
                   ,(embed/url
                     (lambda _
                       ; A new frame has been created
                       (define last (web-cell-ref a-counter))
                       ; We can inspect the value at the parent
```

```
                (web-cell-shadow a-counter (add1 last))
                ; The new frame has been modified
                (loop)))])
          "+")))))))
```

2.7.1 Stateless Web Cells

```
(require web-server/lang/web-cells)
```

The `web-server/lang/web-cells` library provides the same API as `web-server/servlet/web-cells`, but in a way compatible with the Web Language. The one difference is that `make-web-cell` is syntax, rather than a function.

```
(web-cell? v) → boolean?
  v : any/c
(make-web-cell default-expr)
(web-cell-ref wc) → any/c
  wc : web-cell?
(web-cell-shadow wc v) → void
  wc : web-cell?
  v : any/c
```

See `web-server/servlet/web-cells`.

2.8 File Boxes

```
(require web-server/lang/file-box)
```

As mentioned earlier, it is dangerous to rely on the store in Web Language servlets, due to the deployment scenarios available to them. `"lang/file-box.ss"` provides a simple API to replace boxes in a safe way.

```
(file-box? v) → boolean?
  v : any/c
```

Checks if *v* is a file-box.

```
(file-box p v) → file-box?
  p : path-string?
  v : serializable?
```

Creates a file-box that is stored at *p*, with the default contents of *v*.

```
(file-unbox fb) → serializable?
  fb : file-box?
```

Returns the value inside *fb*

```
(file-box-set? fb) → boolean?
  fb : file-box?
```

Returns #t if *fb* contains a value.

```
(file-box-set! fb v) → void
  fb : file-box?
  v : serializable?
```

Saves *v* in the file represented by *fb*.

Warning: If you plan on using a load-balancer, make sure your file-boxes are on a shared medium.

2.9 Stateless Web Parameters

```
(require web-server/lang/web-param)
```

It is not easy to use `parameterize` in the Web Language. `"lang/web-param.ss"` provides (roughly) the same functionality in a way that is serializable. Like other serializable things in the Web Language, they are sensitive to source code modification.

```
(make-web-parameter default)
```

Expands to the definition of a web-parameter with *default* as the default value. A web-parameter is a procedure that, when called with zero arguments, returns *default* or the last value `web-parameterized` in the dynamic context of the call.

```
(web-parameter? v) → boolean?
  v : any/c
```

Checks if *v* appears to be a web-parameter.

```
(web-parameterize ([web-parameter-expr value-expr] ...) expr ...)
```

Runs (begin *expr* ...) such that the web-parameters that the *web-parameter-expr*s evaluate to are bound to the *value-expr*s. From the perspective of the *value-expr*s, this is like let.

2.10 Formlets

```
(require web-server/formlets)
```

The Web Server provides a kind of Web form abstraction called a formlet.

Formlets originate in the work of the Links research group in their paper The Essence of Form Abstraction.

2.10.1 Basic Formlet Usage

Suppose we want to create an abstraction of entering a date in an HTML form. The following formlet captures this idea:

```
(define date-formlet
  (formlet
   (div
    "Month:" ,{input-int . => . month}
    "Day:" ,{input-int . => . day})
   (list month day)))
```

The first part of the formlet syntax is the template of an X-expression that is the rendering of the formlet. It can contain elements like ,(=> *formlet name*) where *formlet* is a formlet expression and *name* is an identifier bound in the second part of the formlet syntax.

This formlet is displayed (with formlet-display) as the following X-expression forest (list):

```
(list
 '(div "Month:" (input ([name "input_0"]))
       "Day:" (input ([name "input_1"])))))
```

date-formlet not only captures the rendering of the form, but also the request processing logic. If we send it an HTTP request with bindings for "input_0" to "10" and "input_1" to "3", with formlet-process, then it returns:

```
(list 10 3)
```

which is the second part of the formlet syntax, where month has been replaced with the integer represented by the "input_0" and day has been replaced with the integer represented by the "input_1".

The real power of formlet is that they can be embedded within one another. For instance, suppose we want to combine two date forms to capture a travel itinerary. The following formlet does the job:

```
(define travel-formlet
  (formlet
   (div
    "Name:" ,{input-string . => . name}
    (div
     "Arrive:" ,{date-formlet . => . arrive}
     "Depart:" ,{date-formlet . => . depart})
    (list name arrive depart))))
```

(Notice that `date-formlet` is embedded twice.) This is rendered as:

```
(list
 '(div
   "Name:"
   (input ([name "input_0"]))
   (div
    "Arrive:"
    (div "Month:" (input ([name "input_1"]))
         "Day:" (input ([name "input_2"])))
    "Depart:"
    (div "Month:" (input ([name "input_3"]))
         "Day:" (input ([name "input_4"]))))))
```

Observe that `formlet-display` has automatically generated unique names for each input element. When we pass bindings for these names to `formlet-process`, the following list is returned:

```
(list "Jay"
      (list 10 3)
      (list 10 6))
```

The rest of the manual gives the details of formlet usage and extension.

2.10.2 Syntactic Shorthand

```
(require web-server/formlets/syntax)
```

Most users will want to use the syntactic shorthand for creating formlets.

`(formlet` *rendering yields-expr*`)`

Constructs a formlet with the specified *rendering* and the processing resulting in the *yields-expr* expression. The *rendering* form is a quasiquoted X-expression, with two special caveats:

,{=> *formlet-expr name*} embeds the formlet given by *formlet-expr*; the result of this processing this formlet is available in the *yields-expr* as *name*.

(#%# *xexpr* ...) renders an X-expression forest.

2.10.3 Functional Usage

```
(require web-server/formlets/lib)
```

The syntactic shorthand abbreviates the construction of *formlet*s with the following library. These combinators may be used directly to construct low-level formlets, such as those for new INPUT element types. Refer to §2.10.4 "Predefined Formlets" for example low-level formlets using these combinators.

---

xexpr-forest/c : contract?

Equivalent to (listof xexpr?)

(formlet/c *content*) → contract?
  content : any/c

Equivalent to (-> integer? (values xexpr-forest/c (-> (listof binding?) (coerce-contract 'formlet/c *content*)) integer?)).

A formlet's internal representation is a function from an initial input number to an X-expression forest rendering, a processing function, and the next allowable input number.

(pure *value*) → (formlet/c any/c)
  value : any/c

Constructs a formlet that has no rendering and always returns *value* in the processing stage.

(cross *f g*) → (formlet/c any/c)
  f : (formlet/c (any/c . -> . any/c))
  g : (formlet/c any/c)

Constructs a formlet with a rendering equal to the concatenation of the renderings of formlets *f* and *g*; a processing stage that applies *g*'s processing result to *f*'s processing result.

```
(cross* f g ...) → (formlet/c any/c)
  f : (formlet/c (() () #:rest (listof any/c) . ->* . any/c))
  g : (formlet/c any/c)
```

Equivalent to `cross` lifted to many arguments.

```
(xml-forest r) → (formlet/c procedure?)
  r : xexpr-forest/c
```

Constructs a formlet with the rendering `r` and the identity procedure as the processing step.

```
(xml r) → (formlet/c procedure?)
  r : xexpr?
```

Equivalent to `(xml-forest (list r))`.

```
(text r) → (formlet/c procedure?)
  r : string?
```

Equivalent to `(xml r)`.

```
(tag-xexpr tag attrs inner) → (formlet/c any/c)
  tag : symbol?
  attrs : (listof (list/c symbol? string?))
  inner : (formlet/c any/c)
```

Constructs a formlet with the rendering `(list (list* tag attrs inner-rendering))` where `inner-rendering` is the rendering of `inner` and the processing stage identical to `inner`.

```
(formlet-display f) → xexpr-forest/c
  f : (formlet/c any/c)
```

Renders `f`.

```
(formlet-process f r) → any/c
  f : (formlet/c any/c)
  r : request?
```

Runs the processing stage of `f` on the bindings in `r`.

2.10.4 Predefined Formlets

```
(require web-server/formlets/input)
```

There are a few basic formlets provided by this library.

`input-string` : `(formlet/c string?)`

A formlet that renders as

```
(list `(input ([name (format "input_~a" next-id)])))
```

where *next-id* is the next available input index and extracts `(format "input_~a" next-id)` in the processing stage and converts it into a UTF-8 string.

`input-int` : `(formlet/c integer?)`

Equivalent to `(cross (pure string->number) input-string)`.

`input-symbol` : `(formlet/c symbol?)`

Equivalent to `(cross (pure string->symbol) input-string)`.

2.10.5 Utilities

```
(require web-server/formlets/servlet)
```

A few utilities are provided for using formlets in Web applications.

```
(send/formlet f [#:wrap wrapper]) → any/c
  f : (formlet/c any/c)
  wrapper : (xexpr? . -> . response?)
          = (lambda (form-xexpr)
              `(html (head (title "Form Entry"))
                     (body ,form-xexpr)))
```

Uses `send/suspend` to send *f*'s rendering (wrapped in a FORM tag whose action is the continuation URL (wrapped again by *wrapper*)) to the client. When the form is submitted, the request is passed to the processing stage of *f*.

```
(embed-formlet embed/url f) → xexpr?
```

```
embed/url : embed/url/c
f : (formlet/c any/c)
```

Like send/formlet, but for use with send/suspend/dispatch.

2.11 Templates

```
(require web-server/templates)
```

The Web Server provides a powerful Web template system for separating the presentation logic of a Web application and enabling non-programmers to contribute to PLT-based Web applications.

Although all the examples here generate HTML, the template language and the §15 "Text Preprocessor" it is based on can be used to generate any text-based format: C, SQL, form emails, reports, etc.

2.11.1 Static

Suppose we have a file "static.html" with the contents:

```
<html>
 <head><title>Fastest Templates in the West!</title></head>
 <body>
  <h1>Bang!</h1>
  <h2>Bang!</h2>
 </body>
</html>
```

If we write the following in our code:

```
(include-template "static.html")
```

Then the contents of "static.html" will be read *at compile time* and compiled into a Scheme program that returns the contents of "static.html" as a string:

```
"<html>\n  <head><title>Fastest Templates in the
West!</title></head>\n  <body>\n    <h1>Bang!</h1>\n    <h2>Bang!</h2>\n  </body>\n</html>"
```

2.11.2 Dynamic

include-template gives the template access to the *complete lexical context* of the including program. This context can be accessed via the §3 "@-Reader" syntax. For example, if "simple.html" contains:

```
<html>
```

```
 <head><title>Fastest @thing in the West!</title></head>
 <body>
  <h1>Bang!</h1>
  <h2>Bang!</h2>
 </body>
</html>
```

Then

```
(let ([thing "Templates"])
  (include-template "simple.html"))
```

evaluates to the same content as the static example.

There are no constraints on how the lexical context of the template is populated. For instance, you can built template abstractions by wrapping the inclusion of a template in a function:

```
(define (fast-template thing)
  (include-template "simple.html"))

(fast-template "Templates")
(fast-template "Noodles")
```

evalutes to two strings with the predictable contents:

```
<html>
 <head><title>Fastest Templates in the West!</title></head>
 <body>
  <h1>Bang!</h1>
  <h2>Bang!</h2>
 </body>
</html>
```

and

```
<html>
 <head><title>Fastest Noodles in the West!</title></head>
 <body>
  <h1>Bang!</h1>
  <h2>Bang!</h2>
 </body>
</html>
```

Furthermore, there are no constraints on the Scheme used by templates: they can use macros, structs, continuation marks, threads, etc. However, Scheme values that are ultimately returned must be printable by the §15 “Text Preprocessor”. For example, consider the following outputs of the title line of different calls to `fast-template`:

- `(fast-template 'Templates)`

```
<head><title>Fastest Templates in the West!</title></head>
```

- `(fast-template 42)`

```
<head><title>Fastest 42 in the West!</title></head>
```

- `(fast-template (list "Noo" "dles"))`

```
<head><title>Fastest Noodles in the West!</title></head>
```

- `(fast-template (lambda () "Thunks"))`

```
<head><title>Fastest Thunks in the West!</title></head>
```

- `(fast-template (delay "Laziness"))`

```
<head><title>Fastest Laziness in the West!</title></head>
```

2.11.3 Gotchas

To obtain an @ symbol in template output, you must escape the @ symbol, because it is the escape character of the §3 "@-Reader" syntax. For example, to obtain:

```
<head><title>Fastest @s in the West!</title></head>
```

You must write:

```
<head><title>Fastest @"@"s in the West!</title></head>
```

as your template: literal @s must be replaced with @"@".

The other gotcha is that since the template is compiled into a Scheme program, only its results will be printed. For example, suppose we have the template:

```
<table>
 @for[([c clients])]{
  <tr><td>@(car c), @(cdr c)</td></tr>
 }
</table>
```

If this is included in a lexical context with `clients` bound to

```
(list (cons "Young" "Brigham") (cons "Smith" "Joseph"))
```

then the template will be printed as:

```
<table>
</table>
```

because `for` does not return the value of the body. Suppose that we change the template to use `for/list` (which combines them into a list):

```
<table>
 @for/list[([c clients])]{
  <tr><td>@(car c), @(cdr c)</td></tr>
 }
</table>
```

Now the result is:

```
<table>
 </tr>
 </tr>
</table>
```

because only the final expression of the body of the `for/list` is included in the result. We can capture all the sub-expressions by using `list` in the body:

```
<table>
 @for/list[([c clients])]{
  @list{
   <tr><td>@(car c), @(cdr c)</td></tr>
  }
 }
</table>
```

Now the result is:

```
<table>
 <tr><td>Young, Brigham</td></tr>
 <tr><td>Smith, Joseph</td></tr>
</table>
```

The templating library provides a syntactic form to deal with this issue for you called `in`:

```
<table>
 @in[c clients]{
  <tr><td>@(car c), @(cdr c)</td></tr>
 }
</table>
```

Notice how it also avoids the absurd amount of punctuation on line two.

2.11.4 HTTP Responses

The quickest way to generate an HTTP response from a template is using the `list` response type:

```
(list #"text/html" (include-template "static.html"))
```

If you want more control then you can generate a `response/full` struct:

```
(make-response/full
 200 "Okay"
 (current-seconds) TEXT/HTML-MIME-TYPE
 empty
 (list (include-template "static.html")))
```

Finally, if you want to include the contents of a template inside a larger X-expression :

```
`(html ,(include-template "static.html"))
```

will result in the literal string being included (and entity-escaped). If you actually want the template to be unescaped, then create a `cdata` structure:

```
`(html ,(make-cdata #f #f (include-template "static.html")))
```

2.11.5 API Details

`(include-template` *path*`)`

Compiles the template at *path* using the §3 "@-Reader" syntax within the enclosing lexical context.

Example:

```
(include-template "static.html")
```

`(in` *x xs e ...*`)`

Expands into

```
(for/list ([x xs])
 (begin/text e ...))
```

Template Example:

```
@in[c clients]{
 <tr><td>@(car c), @(cdr c)</td></tr>
}
```

Scheme Example:

```
(in c clients "<tr><td>" (car c) ", " (cdr c) "</td></tr>")
```

2.11.6 Conversion Example

Al Church has been maintaining a blog with PLT Scheme for some years and would like to convert to `web-server/templates`.

The data-structures he uses are defined as:

```
(define-struct post (title body))

(define posts
  (list
   (make-post
    "(Y Y) Works: The Why of Y"
    "Why is Y, that is the question.")
   (make-post
    "Church and the States"
    "As you may know, I grew up in DC, not technically a state.")))
```

Actually, Al Church-encodes these posts, but for explanatory reasons, we'll use structs.

He has divided his code into presentation functions and logic functions. We'll look at the presentation functions first.

The first presentation function defines the common layout of all pages.

```
(define (template section body)
  `(html
    (head (title "Al's Church: " ,section))
    (body
     (h1 "Al's Church: " ,section)
     (div ([id "main"])
          ,@body))))
```

One of the things to notice here is the `unquote-splicing` on the `body` argument. This indicates that the `body` is list of X-expressions. If he had accidentally used only `unquote` then there would be an error in converting the return value to an HTTP response.

```
(define (blog-posted title body k-url)
```

```
  `((h2 ,title)
    (p ,body)
    (h1 (a ([href ,k-url]) "Continue"))))
```

Here's an example of simple body that uses a list of X-expressions to show the newly posted blog entry, before continuing to redisplay the main page. Let's look at a more complicated body:

```
(define (blog-posts k-url)
  (append
   (apply append
          (for/list ([p posts])
            `((h2 ,(post-title p))
              (p ,(post-body p)))))
   `((h1 "New Post")
     (form ([action ,k-url])
           (input ([name "title"]))
           (input ([name "body"]))
           (input ([type "submit"]))))))
```

This function shows a number of common patterns that are required by X-expressions. First, `append` is used to combine different X-expression lists. Second, `apply append` is used to collapse and combine the results of a `for/list` where each iteration results in a list of X-expressions. We'll see that these patterns are unnecessary with templates. Another annoying patterns shows up when Al tries to add CSS styling and some JavaScript from Google Analytics to all the pages of his blog. He changes the `template` function to:

```
(define (template section body)
  `(html
    (head
     (title "Al's Church: " ,section)
     (style ([type "text/css"])
            "body {margin: 0px; padding: 10px;}"
            "#main {background: #dddddd;}"))
    (body
     (script
      ([type "text/javascript"])
      ,(make-cdata
        #f #f
        "var gaJsHost = ((\"https:\" =="
        "document.location.protocol)"
        "? \"https://ssl.\" : \"http://www.\");"
        "document.write(unescape(\"%3Cscript src='\" + gaJsHost"
        "+ \"google-analytics.com/ga.js' "
        "type='text/javascript'%3E%3C/script%3E\"));"))
     (script
      ([type "text/javascript"])
```

```
 ,(make-cdata
   #f #f
   "var pageTracker = _gat._getTracker(\"UA-YYYYYYY-Y\");"
   "pageTracker._trackPageview();"))
(h1 "Al's Church: " ,section)
(div ([id "main"])
     ,@body))))
```

Some of these problems go away by using here strings, as described in the documentation on §12.6.6 "Reading Strings".

The first thing we notice is that encoding CSS as a string is rather primitive. Encoding JavaScript with strings is even worse for two reasons: first, we are more likely to need to manually escape characters such as "; second, we need to use a CDATA object, because most JavaScript code uses characters that "need" to be escaped in XML, such as &, but most browsers will fail if these characters are entity-encoded. These are all problems that go away with templates.

Before moving to templates, let's look at the logic functions:

```
(define (extract-post req)
  (define binds
    (request-bindings req))
  (define title
    (extract-binding/single 'title binds))
  (define body
    (extract-binding/single 'body binds))
  (set! posts
        (list* (make-post title body)
               posts))
  (send/suspend
   (lambda (k-url)
     (template "Posted" (blog-posted title body k-url))))
  (display-posts))

(define (display-posts)
  (extract-post
   (send/suspend
    (lambda (k-url)
      (template "Posts" (blog-posts k-url))))))

(define (start req)
  (display-posts))
```

To use templates, we need only change `template`, `blog-posted`, and `blog-posts`:

```
(define (template section body)
  (list TEXT/HTML-MIME-TYPE
        (include-template "blog.html")))
```

```
(define (blog-posted title body k-url)
  (include-template "blog-posted.html"))

(define (blog-posts k-url)
  (include-template "blog-posts.html"))
```

Each of the templates are given below:

"blog.html":

```
<html>
 <head>
  <title>Al's Church: @|section|</title>
  <style type="text/css">
   body {
    margin: 0px;
    padding: 10px;
   }

   #main {
    background: #dddddd;
   }
  </style>
 </head>
 <body>
  <script type="text/javascript">
   var gaJsHost = (("https:" == document.location.protocol) ?
     "https://ssl." : "http://www.");
   document.write(unescape("%3Cscript src='" + gaJsHost +
     "google-analytics.com/ga.js'
      type='text/javascript'%3E%3C/script%3E"));
  </script>
  <script type="text/javascript">
   var pageTracker = _gat._getTracker("UA-YYYYYYY-Y");
   pageTracker._trackPageview();
  </script>

  <h1>Al's Church: @|section|</h1>
  <div id="main">
    @body
  </div>
 </body>
</html>
```

Notice that this part of the presentation is much simpler, because the CSS and JavaScript can be included verbatim, without resorting to any special escape-escaping patterns. Similarly,

since the body is represented as a string, there is no need to remember if splicing is necessary.

```
"blog-posted.html":

  <h2>@|title|</h2>
  <p>@|body|</p>

  <h1><a href="@|k-url|">Continue</a></h1>

"blog-posts.html":

  @in[p posts]{
   <h2>@(post-title p)</h2>
   <p>@(post-body p)</p>
  }

  <h1>New Post</h1>
  <form action="@|k-url|">
   <input name="title" />
   <input name="body" />
   <input type="submit" />
  </form>
```

Compare this template with the original presentation function: there is no need to worry about managing how lists are nested: the defaults *just work*.

2.12 Continuation Managers

Since Scheme servlets store their continuations on the server, they take up memory on the server. Furthermore, garbage collection can not be used to free this memory, because there are roots outside the system: users' browsers, bookmarks, brains, and notebooks. Therefore, some other strategy must be used if memory usage is to be controlled. This functionality is pluggable through the manager interface.

2.12.1 General

```
(require web-server/managers/manager)
```

`"managers/manager.ss"` defines the manager interface. It is required by the users and implementers of managers.

```
(struct manager (create-instance
                 adjust-timeout!
                 clear-continuations!
                 continuation-store!
                 continuation-lookup))
  create-instance : ((-> void) . -> . number?)
  adjust-timeout! : (number? number? . -> . void)
  clear-continuations! : (number? . -> . void)
  continuation-store! : (number? any/c expiration-handler/c . -> . (list/c number? number?))
  continuation-lookup : (number? number? number? . -> . any/c)
```

`create-instance` is called to initialize a instance, to hold the continuations of one servlet session. It is passed a function to call when the instance is expired. It runs the id of the instance.

`adjust-timeout!` is a to-be-deprecated function that takes an instance-id and a number. It is specific to the timeout-based manager and will be removed.

`clear-continuations!` expires all the continuations of an instance.

`continuation-store!` is given an instance-id, a continuation value, and a function to include in the exception thrown if the continuation is looked up and has been expired. The two numbers returned are a continuation-id and a nonce.

`continuation-lookup` finds the continuation value associated with the instance-id, continuation-id, and nonce triple it is given.

```
(struct (exn:fail:servlet-manager:no-instance exn:fail) (expiration-handler
  expiration-handler : expiration-handler/c
```

This exception should be thrown by a manager when an instance is looked up that does not exist.

```
(struct (exn:fail:servlet-manager:no-continuation exn:fail) (expiration-handler
  expiration-handler : expiration-handler/c
```

This exception should be thrown by a manager when a continuation is looked up that does not exist.

2.12.2 No Continuations

```
(require web-server/managers/none)
```

"managers/none.ss" defines a manager constructor:

```
(create-none-manager instance-expiration-handler) → manager?
  instance-expiration-handler : expiration-handler/c
```

This manager does not actually store any continuation or instance data. You could use it if you know your servlet does not use the continuation capturing functions and want the server to not allocate meta-data structures for each instance.

If you *do* use a continuation capturing function, the continuation is simply not stored. If the URL is visited, the *instance-expiration-handler* is called with the request.

If you are considering using this manager, also consider using the Web Language. (See §2.1.3 "Stateless Servlets".)

2.12.3 Timeouts

```
(require web-server/managers/timeouts)
```

`"managers/timeouts.ss"` defines a manager constructor:

```
(create-timeout-manager instance-exp-handler
                        instance-timeout
                        continuation-timeout) → manager?
  instance-exp-handler : expiration-handler/c
  instance-timeout : number?
  continuation-timeout : number?
```

Instances managed by this manager will be expired *instance-timeout* seconds after the last time it is accessed. If an expired instance is looked up, the `exn:fail:servlet-manager:no-instance` exception is thrown with *instance-exp-handler* as the expiration handler.

Continuations managed by this manager will be expired *continuation-timeout* seconds after the last time it is accessed. If an expired continuation is looked up, the `exn:fail:servlet-manager:no-continuation` exception is thrown with *instance-exp-handler* as the expiration handler, if no expiration-handler was passed to `continuation-store!`.

`adjust-timeout!` corresponds to `reset-timer!` on the timer responsible for the servlet instance.

This manager has been found to be... problematic... in large-scale deployments of the Web Server .

2.12.4 LRU

```
(require web-server/managers/lru)
```

"managers/lru.ss" defines a manager constructor:

```
(create-LRU-manager  instance-expiration-handler
                     check-interval
                     collect-interval
                     collect?
                    [#:initial-count initial-count
                     #:inform-p inform-p])          → manager?
  instance-expiration-handler : expiration-handler/c
  check-interval : integer?
  collect-interval : integer?
  collect? : (-> boolean?)
  initial-count : integer? = 1
  inform-p : (integer? . -> . void) = (lambda _ (void))
```

Instances managed by this manager will be expired if there are no continuations associated with them, after the instance is unlocked. If an expired instance is looked up, the exn:fail:servlet-manager:no-instance exception is thrown with instance-exp-handler as the expiration handler.

Continuations managed by this manager are given a "Life Count" of *initial-count* initially. If an expired continuation is looked up, the exn:fail:servlet-manager:no-continuation exception is thrown with instance-exp-handler as the expiration handler, if no expiration-handler was passed to continuation-store!.

Every *check-interval* seconds *collect?* is called to determine if the collection routine should be run. Every *collect-interval* seconds the collection routine is run.

Every time the collection routine runs, the "Life Count" of every continuation is decremented by 1. If a continuation's count reaches 0, it is expired. The *inform-p* function is called if any continuations are expired, with the number of continuations expired.

The recommended usage of this manager is codified as the following function:

```
(make-threshold-LRU-manager instance-expiration-handler
                            memory-threshold)
 → manager?
  instance-expiration-handler : expiration-handler/c
  memory-threshold : number?
```

This creates an LRU manager with the following behavior: The memory limit is set to

memory-threshold bytes. Continuations start with 24 life points. Life points are deducted at the rate of one every 10 minutes, or one every 5 seconds when the memory limit is exceeded. Hence the maximum life time for a continuation is 4 hours, and the minimum is 2 minutes.

If the load on the server spikes—as indicated by memory usage—the server will quickly expire continuations, until the memory is back under control. If the load stays low, it will still efficiently expire old continuations.

3 Extending the Web Server

3.1 Configuration

There are a number of libraries and utilities useful for configuring the Web Server .

3.1.1 Configuration Table Structure

```
(require web-server/configuration/configuration-table-structs)
```

"configuration/configuration-table-structs.ss" provides the following structures that represent a standard configuration (see §3.5 "Web Server Unit") 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 §3.1.2 "Configuration Table".

```
(struct configuration-table (port
                             max-waiting
                             initial-connection-timeout
                             default-host
                             virtual-hosts))
  port : port-number?
  max-waiting : natural-number/c
  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))
  indices : (listof string?)
  log-format : symbol?
  messages : messages?
  timeouts : timeouts?
  paths : paths?
```

```
(struct host (indices
              log-format
              log-path
              passwords
              responders
              timeouts
              paths))
  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))
  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))
  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))
  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))
  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?)
```

3.1.2 Configuration Table

```
(require web-server/configuration/configuration-table)
```

"configuration/configuration-table.ss" 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.

```
  `((port ,integer?)
    (max-waiting ,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 are relative to 'host-root (other than 'host-root obviously.)

Allowable 'log-formats 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*.

3.1.3 Servlet Namespaces

```
(require web-server/configuration/namespace)
```

"configuration/namespace.ss" provides a function to help create the make-servlet-namespace procedure needed by the make functions of "dispatchers/dispatch-servlets.ss" and "dispatchers/dispatch-lang.ss".

---

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 `mzscheme` 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.ss" "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, due to a subtlety in the way MzScheme implements structures.

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 §3.3 "Dispatchers") different sets of servlets can share different sets of modules.

3.1.4 Standard Responders

```
(require web-server/configuration/responders)
```

`"configuration/responders.ss"` provides some functions that help constructing HTTP responders. These functions are used by the default dispatcher constructor (see §3.5 "Web Server Unit") 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/full 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.

```
(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*.

3.2 Setting Up Servlets

```
(require web-server/servlet/setup)
```

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? . -> . 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? . -> . 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 start) → servlet?
  directory : path-string?
  start : (request? . -> . response?)
```

Creates a stateless web-server servlet that uses *directory* as its current directory 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.)

3.2.1 Internal Servlet Representation

```
(require web-server/private/servlet)
```

```
(struct servlet (custodian namespace manager directory handler)
        #:mutable)
  custodian : custodian?
  namespace : namespace?
  manager : manager?
  directory : path-string?
  handler : (request? . -> . 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.

3.3 Dispatchers

The Web Server is really just a particular configuration of a dispatching server. There are a number of dispatchers that are defined to support the Web Server . Other dispatching servers, or variants of the Web Server , may find these useful. In particular, if you want a peculiar processing pipeline for your Web Server installation, this documentation will be useful.

3.3.1 General

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

`"dispatchers/dispatch.ss"` 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 ())`

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))]))))
```

3.3.2 Mapping URLs to Paths

```
(require web-server/dispatchers/filesystem-map)
```

"dispatchers/filesystem-map.ss" provides a means of mapping URLs to paths on the filesystem.

---

url->path/c : contract?

This contract is equivalent to (->* (url?) (path? (listof path-element?))). 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.

3.3.3 Sequencing

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

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.

3.3.4 Timeouts

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

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*.

3.3.5 Lifting Procedures

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

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.

3.3.6 Filtering Requests

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

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.

3.3.7 Procedure Invocation upon Request

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

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.

3.3.8 Logging

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

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.

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

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`.

3.3.9 Password Protection

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

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 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")))
```

3.3.10 Virtual Hosts

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

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.

3.3.11 Serving Files

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

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? . -> . bytes?)
                  = (lambda (path) TEXT/HTML-MIME-TYPE)
  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.

3.3.12 Serving Servlets

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

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? . -> . response?)
                             = servlet-loading-responder
  responders-servlet : (url? exn? . -> . response?)
                     = servlet-error-responder
```

This dispatcher runs Scheme 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.

3.3.13 Statistics

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

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.

3.3.14 Limiting Requests

```
(require web-server/dispatchers/limit)
```

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 scheme

(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
            (make-response/full
             200 "Okay"
             (current-seconds) TEXT/HTML-MIME-TYPE
             empty
             (list (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
           (make-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)
```

3.4 Web Config Unit

The Web Server offers a unit-based approach to configuring the server.

3.4.1 Configuration Signature

```
(require web-server/web-config-sig)
```

`web-config^` : signature

Provides contains the following identifiers.

`max-waiting` : `integer?`

Passed to `tcp-accept`.

`virtual-hosts` : `(listof (cons/c string? host-table?))`

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` : `string?`

Passed to `tcp-accept`.

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

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

3.4.2 Configuration Units

```
(require web-server/web-config-unit)
```

```
(configuration-table->web-config@
 path
 [#:port port
  #:listen-ip listen-ip
  #:make-servlet-namespace make-servlet-namespace])
→ (unit? 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? 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.

3.5 Web Server Unit

The Web Server offers a unit-based approach to running the server.

3.5.1 Signature

```
(require web-server/web-server-sig)
```

`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*.

3.5.2 Unit

```
(require web-server/web-server-unit)
```

```
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.
- Execute servlets in the 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.

3.6 Internal

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.

3.6.1 Timers

```
(require web-server/private/timer)
```

"private/timer.ss" provides a functionality for running procedures after a given amount of time, that may be extended.

```
(struct timer (evt expire-seconds action))
  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) → void
```

Handles the execution and management of timers.

```
(start-timer s action) → timer?
  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*.

3.6.2 Connection Manager

```
(require web-server/private/connection-manager)
```

"private/connection-manager.ss" 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?))
  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.

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

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

```
(new-connection timeout
                i-port
                o-port
                cust
                close?) → connection?
  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 reset-timer! with the timer behind *c* with *t*.

3.6.3 Dispatching Server

The Web Server is just a configuration of a dispatching server. This dispatching server component is useful on its own.

Dispatching Server Signatures

```
(require web-server/private/dispatch-server-sig)
```

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) → (-> 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*.

---

dispatch-server-config^ : signature

---

port : port?

Specifies the port to serve on.

---

listen-ip : string?

Passed to tcp-accept.

---

max-waiting : integer?

Passed to tcp-accept.

---

initial-connection-timeout : integer?

Specifies the initial timeout given to a connection.

```
(read-request c p port-addresses) → any/c
  c : connection?
  p : port?
  port-addresses : (-> port? boolean?
                       (or/c (values string? string?)
                             (values string? (integer-in 1 65535)
                                     string? (integer-in 1 65535))))
```

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

---

dispatch : dispatcher/c

How to handle requests.

Dispatching Server Unit

```
(require web-server/private/dispatch-server-unit)
```

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

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

Runs the dispatching server config in a very basic way, except that it uses §3.6.2 “Connection Manager” to manage connections.

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.

3.6.4 Serializable Closures

```
(require web-server/private/closure)
```

The defunctionalization process of the Web Language (see §2.1.3 "Stateless Servlets") requires an explicit representation of closures that is serializable. `"private/closure.ss"` is this representation. It provides:

```
(make-closure-definition-syntax tag
                                fvars
                                proc) → syntax?
  tag : syntax?
  fvars : (listof identifier?)
  proc : syntax?
```

Outputs a syntax object that defines a serializable structure, with *tag* as the tag, that represents a closure over *fvars*, that acts a procedure and when invoked calls *proc*, which is assumed to be syntax of `lambda` or `case-lambda`.

```
(closure->deserialize-name c) → symbol?
  c : closure?
```

Extracts the unique tag of a closure *c*.

These are difficult to use directly, so `"private/define-closure.ss"` defines a helper form:

Define Closure

```
(require web-server/private/define-closure)
```

```
(define-closure tag formals (free-vars ...) body)
```

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

Here is an example:

```
#lang scheme
(require scheme/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
```

3.6.5 Cache Table

```
(require web-server/private/cache-table)
```

"private/cache-table.ss" 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.

3.6.6 MIME Types

```
(require web-server/private/mime-types)
```

"private/mime-types.ss" provides function for dealing with "mime.types" files.

```
(read-mime-types p) → (hash-table/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? . -> . bytes?)
  p : path-string?
```

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

3.6.7 Serialization Utilities

```
(require web-server/private/mod-map)
```

The scheme/serialize library provides the functionality of serializing values. "private/mod-map.ss" 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*.

3.6.8 URL Param

```
(require web-server/private/url-param)
```

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. "private/url-param.ss" 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.

3.6.9 Miscellaneous Utilities

```
(require web-server/private/util)
```

There are a number of other miscellaneous utilities the Web Server needs. They are provided by "private/util.ss".

Contracts

```
port-number? : contract?
```

Equivalent to `(between/c 1 65535)`.

```
path-element? : contract?
```

Equivalent to `(or/c path-string? (symbols 'up 'same))`.

Lists

```
(list-prefix? l r) → boolean?
  l : list?
  r : list?
```

True if *l* is a prefix of *r*.

URLs

```
(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.

Paths

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

Like `normalize-path`, but does not resolve symlinks.

```
(path-without-base base p) → (listof path-element?)
  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.

```
(strip-prefix-ups p) → (listof path-element?)
  p : (listof path-element?)
```

Removes all the prefix `".."`s from *p*.

Exceptions

```
(pretty-print-invalid-xexpr exn v) → void
  exn : exn:invalid-xexpr?
  v : any/c
```

Prints *v* as if it were almost an X-expression highlighting the error according to *exn*.

```
(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.

Strings

```
(lowercase-symbol! sb) → symbol?
  sb : (or/c string? bytes?)
```

Returns *sb* as a lowercase symbol.

```
(read/string s) → serializable?
  s : string?
```

reads a value from *s* and returns it.

```
(write/string v) → string?
  v : serializable?
```

writes *v* to a string and returns it.

Bytes

```
(read/bytes b) → serializable?
  b : bytes?
```

reads a value from *b* and returns it.

```
(write/bytes v) → bytes?
  v : serializable?
```

writes *v* to a bytes and returns it.

4 Troubleshooting and Tips

4.1 Why are my servlets not updating on the server when I change the code on disk?

By default, the server uses `make-cached-url->servlet` to load servlets from the disk. As it loads them, they are cached and the disk is not referred to for future requests. This ensures that there is a single namespace for each servlet, so that different instances can share resources, such as database connections, and communicate through the store. The default configuration of the server (meaning the dispatcher sequence used when you load a configuration file) provides a special URL to localhost that will reset the cache: `"/conf/refresh-servlets"`. If you want the server to reload your changed servlet code, then GET this URL and the server will reload the servlet on the next request.

4.2 What special considerations are there for security with the Web Server?

The biggest problem is that a naive usage of continuations will allow continuations to subvert authentication mechanisms. Typically, all that is necessary to execute a continuation is its URL. Thus, URLs must be as protected as the information in the continuation.

Consider if you link to a public site from a private continuation URL: the `Referrer` field in the new HTTP request will contain the private URL. Furthermore, if your HTTP traffic is in the clear, then these URLs can be easily poached.

One solution to this is to use a special cookie as an authenticator. This way, if a URL escapes, it will not be able to be used, unless the cookie is present. For advice about how to do this well, see Dos and Don'ts of Client Authentication on the Web from the MIT Cookie Eaters.

Note: It may be considered a great feature that URLs can be shared this way, because delegation is easily built into an application via URLs.

4.3 How do I use Apache with the PLT Web Server?

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

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

The first argument to `RewriteRule` is a match pattern. The second is how to rewrite the URL. 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 should make a second request.

See Apache's documentation for more details on RewriteRule.

4.4 IE ignores my CSS or behaves strange in other ways

In quirks mode, IE does not parse your page as XML, in particular it will not recognize many instances of "empty tag shorthand", e.g. "", whereas the Web Server uses `xml` to format XML, which uses empty tag shorthand by default. You can change the default with the `empty-tag-shorthand` parameter: `(empty-tag-shorthand 'never)`.

4.5 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 Scheme start-up script, you can write:

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

4.6 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.

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

Refer to §3.3.14 "Limiting Requests".

5 Acknowledgements

We thank Matthew Flatt for his superlative work on MzScheme. We thank the previous maintainers of the Web Server : Paul T. Graunke, Mike Burns, and Greg Pettyjohn Numerous people have provided invaluable feedback on the server, including Eli Barzilay, Ryan Culpepper, Robby Findler, Dave Gurnell, Matt Jadud, Dan Licata, Jacob Matthews, Matthias Radestock, Andrey Skylar, Michael Sperber, Anton van Straaten, Dave Tucker, and Noel Welsh. We also thank the many other PLT Scheme users who have exercised the server and offered critiques.

Index