CHAPTER 21
Serving HTTP

Socket Servers

Proof of Concept

Let's start with a short example. I'm running SBCL on Ubuntu Linux but this will work as-is on any platform and on any of the nine Lisp implementations (not GCL or Corman) supported by the ASDF-INSTALLable socket interface *usocket*.

```
(defun one-shot-server (port)
  (let ((socket (usocket:socket-listen usocket:*wildcard-host*
                                       port
                                       :reuse-address t)))
    (usocket:wait-for-input socket)
    (let ((stream (usocket:socket-stream (usocket:socket-accept socket))))
      (handle-request stream)
      (close stream)
      (usocket:socket-close socket))))

(defun handle-request (stream)
  (let ((line (read-line stream)))
    (format stream "You said: ~a" line))
  (terpri stream)
  (force-output stream))
```

Now run `(one-shot-server 4567)` in your listener and talk to it from a separate shell:

```
ndl@vulture:~$ telnet localhost 4567
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
beam me up!
You said: beam me up!
Connection closed by foreign host.
ndl@vulture:~$
```


Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Here we have a quick demonstration of a TCP server written in Lisp. We perform some usocket magic and are handed a bidirectional stream; we read data from the client over this stream and send output back.

Make sure you emit a final newline and flush the contents of the stream.

The client itself can be written in Lisp too and is even more straightforward than the server:

```
CL-USER> (defun simple-test (port string)
           (let* ((socket (usocket:socket-connect #(127 0 0 1) port))
                  (stream (usocket:socket-stream socket)))
             (write-line string stream)
             (force-output stream)
             (let ((result (read-line stream)))
               (close stream)
               (usocket:socket-close socket)
               result)))
SIMPLE-TEST
CL-USER> (simple-test 4567 "testing: 1 2 3")
"You said: testing: 1 2 3"
CL-USER>
```

A simple conversation! With a bit of dressing up, code like this could serve... well, any socket protocol you like. However as servers go this example isn't good for much and has some very obvious failings.

Exercise

Start listing them.

We're going to address these issues in the context of building web servers in Lisp. We'll start by expanding the example above, to construct in about 200 lines of code the foundations of a server which sits behind an Apache module called *mod_lisp* and is capable of generating dynamic responses to any request. We'll be playing with closures (Chapter 6) and parts of the condition system Chapter 12). We'll assume some familiarity with both HTTP and the Unix `curl` utility which we'll be using to test things out. In the following three chapters we'll consider web servers in greater detail: parsing input, generating output, and finally the high-level, fully-featured HTML server *Hunchentoot*.

The messages of these chapters are that:

Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

- Lisp is fine for working with socket-based applications (for some reason that I can't explain, rumour has it that this is not the case); and
- by placing Lisp at the heart of your web server you can choose to run arbitrary and powerful code, seamlessly, over every request.

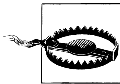
Looping and Threading

The most obvious fault with one-shot-server is that, as the name implies, it's only good for serving a single request. Let's correct that. While we're at it, we'll make the server multi-threaded so it can serve more than one request at a time. Once we've done that we'll have a simple architecture which layers string manipulation on top of two-way streams, threads, and sockets. All your web sever has to do is be intelligent about the strings.

Now's a good time to add some basic cleanups so that if one request or indeed the whole server dies then their resources won't be tied up permanently. We still don't have: error handling, logging, or any knowledge of either HTTP or HTML. We'll graft all these on later.

Rather than use SBCL's MP functionality directly, we'll mimic Hunchentoot and work with the cross-platform (GCL excepted) ASDF-INSTALLable interface *Bordeaux Threads* (installation keyword `:bordeaux-threads`). If you've read Chapter 15 you won't have any problem following my use of this library even though the invocations differ from Allegro's MP. For chapter and verse documentation, visit:

http://trac.common-lisp.net/bordeaux-threads/wiki/ApiDocumentation



SBCL does not support multithreading on Windows. The code below is conditionalized to take account of this.

By default, SBCL is not built with multithreading capabilities on any platform. Fix this by placing a form equivalent to the following in *customize-target-features.lisp* the installation directory, before starting the build:

```
(lambda (list)
  (pushnew :sb-thread list)
  list)
```

See the section on "Customizing SBCL" in the INSTALL file for further details.

```
;; Utility: if Bordeaux Threads are available make the function call in a
;; separate process and return from try-make-thread immediately.

(defun try-make-thread (name function)
  #+bordeaux-threads
```


Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

```
  (bt:make-thread function :name name)
  #-bordeaux-threads
  (funcall function))

;; Create a passive/listening socket and pass this to run-server. Guarantee
;; that the socket will be released when the server halts.

(defvar *server* nil)

(defun start-server (port)
  (let ((socket (usocket:socket-listen usocket:*wildcard-host*
                                       port
                                       :reuse-address t)))
    (setf *server*
          (try-make-thread (format nil "Port ~a server" port)
                           (lambda ()
                             (unwind-protect
                                  (run-server socket)
                               (usocket:socket-close socket)))))))

;; To stop the server we kill its process. The unwind-protect above ensures
;; that the socket will be closed.

#+bordeaux-threads
(defun stop-server ()
  (let ((server (shiftf *server* nil)))
    (when server
      (bt:destroy-thread server))))

;; Loop around, waiting for incoming connections. Each time one arrives,
;; call usocket:socket-stream to create a bidirectional stream and pass
;; this to handle-request, asynchronously if possible. Guarantee that the
;; stream will be closed when handle-request exits. For now, just call the
;; simple handler we defined earlier. We'll redefine that later.

(defun run-server (socket)
  (loop
     (usocket:wait-for-input socket)
     (let ((stream (usocket:socket-stream (usocket:socket-accept socket))))
       (try-make-thread (format nil "Request handler for ~s" stream)
                        (lambda ()
                          (with-open-stream (stream stream)
                            (handle-request stream)))))))
```

Let's try this out:

```
CL-USER> (start-server 4567)
#<SB-THREAD:THREAD "Port 4567 server" RUNNING {ADE4F39}>
CL-USER> (simple-test "Hello")
"You said: Hello"
CL-USER> (simple-test "Goodbye")
"You said: Goodbye"
CL-USER>
```


Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Exercise

Look up the macro `cl:with-open-stream`. What would be disasterously wrong with the following rearrangement?

```
(let ((stream (usocket:socket-stream (usocket:socket-accept socket))))
  (with-open-stream (stream stream)
    (bt:make-thread (lambda ()
                      (handle-request stream)))))
```

Exercise

Without implementing anything complicated, redefine `handle-request` to be a bit more interesting.

An HTTP Proxy

If we want to serve web pages then we will need an implementation of HTTP. The naive approach would be to print off its specification and get work on making handle-request follow it. That's a non-trivial task and there's no good reason to devote any energy to this yourself. If you want to see what the result looks like—HTTP in Lisp from the ground up—skip ahead to Chapter 24. For now we'll see what happens when you hand full-blown HTTP over to Apache and leave your application with one less task to worry about.

We will configure Apache to pass certain requests to your application via the *mod_lisp* plug-in module. Communication between mod_lisp and your application is over a socket, with a protocol much simpler than HTTP. Unlike a CGI script, which generally terminates after serving each request, a mod_lisp server just keeps on running. This allows it the luxury of being larger and slower to start than a single-shot script, the advantage of running faster than interpreted scripting languages, and the option of keeping state in memory. Also—as in all the examples here;—the mod_lisp socket can be in the unpriviledged range and so running the mod_lisp server as an unpriviledged user is trivial.

I'll assume you already have Apache 2 (*http://httpd.apache.org/*) installed. The home page for mod_lisp is

http://www.fractalconcept.com/asp/mod_lisp

and downloads are served from

http://www.fractalconcept.com:8000/public/open-source/mod_lisp/

All you'll need is the single source file *mod_lisp2.c* (1000 lines of C, covered by a three-clause BSD license). Use the "Apache eXtenSion tool" to compile it and install the binary *mod_lisp2.so*, thus:

```
ndl@vulture:~/chapter-21$ sudo /usr/local/apache2/bin/apxs -ci mod_lisp2.c
```


Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

If you're marooned on Windows without a C compiler there's a ZIP file which you can download. Extract *mod_lisp2.so* and copy it into your Apache modules folder (typically *C:\Program Files\Apache Group\Apache\modules*).

The next step is to configure Apache for mod_lisp by adding the following to its configuration (either in *http.conf* or in some file loaded by an `Include` directive).

1. Load the module:

```
LoadModule lisp_module modules/mod_lisp2.so
```

2. Use the `LispServer` directive to list the host on which your Lisp application is running (typically but not necessarily localhost), the port on which it will listen for mod_lisp requests, and an identification string of your choice (this will be sent to your application as the value for the `"server-id"` header).

```
LispServer 127.0.0.1 4567 "mod_lisp example"
```

3. One or more `<Location>` directives to specify the URLs which are to be handled by mod_lisp:

```
<Location /test>
  SetHandler lisp-handler
</Location>
```

4. If you use mod_ssl, add the following to your SSL directives:

```
SSLOptions +StdEnvVars
```

Now restart Apache:

```
ndl@vulture:~/chapter-21$ sudo /usr/local/apache2/bin/apachectl graceful
ndl@vulture:~/chapter-21$
```

Test your configuration. As you're not serving mod_lisp requests on port 4567 yet, you expect to get a "500 Internal Server Error" (and in particular not "404 Not Found").

```
ndl@vulture:~/chapter-21$ curl -I http://vulture/test
HTTP/1.1 500 Internal Server Error
Date: Thu, 19 Nov 2009 11:54:06 GMT
Server: Apache/2.2.14 (Unix) mod_lisp2/1.3.2
Connection: close
Content-Type: text/html; charset=iso-8859-1

ndl@vulture:~/chapter-21$
```

Finally, rewrite `handle-request` to do the absolute minimum work required to verify that mod_lisp is working:

```
(defun handle-request (stream)
  (let ((reply (reply-for-request stream)))
    (write-line "end" stream)
    (write-line reply stream))
  (force-output stream))

(defun reply-for-request (stream)
```


Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

```
  (read-line stream)       ; for now, read and discard first line of input
  (format nil "Hello from mod_lisp~%"))
```

Note that we don't need to restart the server in order to test this new definition!

```
ndl@vulture:~/chapter-21$ curl http://vulture/test
Hello from mod_lisp
ndl@vulture:~/chapter-21$
```

Using mod_lisp

The protocol for exchanging information between mod_lisp's client (Apache) and server (your Lisp session) is really straightforward.

- mod_lisp opens a socket connection to your server.
- Headers and connection parameters are sent as key / value pairs: the key as a string on one line, the value as a string on the next. The input is terminated by the string `"end"` followed by a final newline.
- Headers include `"method"` (value typically either `"GET"` or `"POST"`) and `"url"` (as received by Apache, the part of the URL after the hostname).
- GET parameters can be parsed from the `"url"` header (Chapter 22).
- POST requests have a `"content-length"` header whose value can be parsed as an integer. Read that number of bytes from the stream (after the `"end"` line) to retrieve the POST parameters.
- In both cases parameter values are *application/x-www-form-urlencoded*: spaces become `#\+` characters, non-alphanumerics are replaced by `#\%` followed by their `char-code` as a two-digit number in base 16 (so `#\%` is written as `"%25"`), and newlines are represented by `"%0D%0A"`.
- Output back to the client follows the same pattern: header keys and values on alternating lines, terminated by `"end"` and a fresh line, followed by content.
- If `"content-length"` is sent back then that number of characters must be available as content.
- Send `"Keep-Socket" "1"` to keep the mod_lisp socket open.

We can redefine `handle-request` to process a subset of mod_lisp requests (no POST, no HTML, no failures), thus:

```
(defun handle-request (stream)
  (let* ((headers (parse-modlisp-headers stream))
         (response (generate-response stream headers)))
     (write-response stream response)))

(defun parse-modlisp-headers (stream)
  (loop for key = (read-line stream nil)
        while (and key (string-not-equal key "end"))
        for value = (read-line stream nil)
        collect (keywordify key) collect value))
```


Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

```
(defun keywordify (stringable)
  (intern (string-upcase stringable) :keyword))

(defparameter *response-formatter* "~@{~:(~a~)~%~a~%~}end~%")

(defun write-response (stream response)
  (format stream *response-formatter*
          :status 200
          :content-type "text/plain"
          :content-length (length response))
  (write-string response stream)
  (force-output stream))
```

To see what a complete set of mod_lisp headers looks like, try out the server as defined above with content generated thus:

```
(defun generate-response (stream headers)
  (with-output-to-string (output)
    (pprint headers output)))

ndl@vulture:~/chapter-21$ curl http://localhost/test

(:SERVER-PROTOCOL "HTTP/1.1" :METHOD "GET" :URL "/test"
 :SERVER-IP-ADDR "::1" :SERVER-IP-PORT "80" ...)
ndl@vulture:~/chapter-21$
```

Flesh it Out

Now that we've got the basic framework in place, we can start thinking about using it to build a demonstration web site. We'll keep this as simple as possible for the time being and postpone parsing techniques for Chapter 22 and HTML generation for Chapter 23. This means that we'll temporarily be limited when it comes to picking requests apart, and that any dynamically created responses will be plain text. On the other hand we'll be in a position to

- perform simple content lookup, handle errors, and generate as much output on the fly as we choose; and
- demonstrate four important routes through the server: pages which are served successfully, pages which couldn't be found, requests which resulted in a server error, and—be the friend of webmasters everywhere—redirects.

Writing Responses

We need to redefine just three functions from the listings earlier in the chapter. Let's start by giving `write-response` two additional parameters; it is now sufficiently general to handle almost all the output we need sent back to Apache.

```
(defun write-response (stream status response content-type)
  (format stream *response-formatter*
          :status status
```


Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

```
          :content-type (convert-content-type content-type)
          :content-length (length response))
  (when response
    (write-string response stream))
  (force-output stream))
```

`Status` is a number denoting an HTTP status code. We can go a long way with just four values:

```
(defconstant +http-ok+        200)
(defconstant +http-found+     302)        ; temporary redirect
(defconstant +http-not-found+ 404)
(defconstant +http-error+     500)
```

Because we're confining ourselves to text responses at present, callers of `write-response` will set `content-type` to `:text` and the corresponding Content-Type header is `"text/plain"`:

```
(defmethod convert-content-type ((content-type (eql :text)))
  "text/plain")
```

By using a CLOS method here, we can envisage an upgrade route when we get around to generating HTML; all we'll have to do to fix `write-response` is add a method to `convert-content-type`.

There's one circumstance for which `write-response` doesn't generate the right set of headers, and it's easily addressed. We'll come back to redirection later.

```
(defun write-redirect-response (stream to-url)
  (format stream *response-formatter*
          :status +http-found+
          :location to-url)
  (force-output stream))
```

Fully Featured Request Handler

Here is our final version of `handle-request`:

```
(defun handle-request (stream)
  (let* ((headers (parse-modlisp-headers stream))
         (what (catch 'server-response
                 (handler-bind ((serious-condition
                                 (lambda (error)
                                   (handle-error error))))
                   (multiple-value-bind (response content-type)
                       (generate-response stream headers)
                     (if response
                         (write-response stream
                                         +http-ok+ response content-type)
                       (file-not-found-response stream headers)))
                   nil))))
    (postprocess-response stream what headers)))

(defmethod postprocess-response (stream what headers)
```


Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

```
  (declare (ignore stream what headers))
  nil)
```

Exercise

Compare this with the previous definition of `handle-request` and verify how its three actions (which we might dub: parse, generate, write) are still performed. "All being well," as my grandmother used to say.

Let's pull this function apart in detail.

- As before, the stream and headers are passed to `generate-response`. The stream is included only because if this is a POST request then we'll need to parse parameters from it. Output is returned from this function as a string, along with a keyword denoting content-type; these will be passed to `write-response` which we defined above.
- If a response can't be found then `generate-response` returns `nil` and we call `file-not-found-response` to create a complaint and dispatch that to the stream (by calling `write-response`, as it turns out).
- In either situation, once the output has been generated and dispatched our job is done.
- We can bypass `write-response` by throwing to `server-response`. For example, we might throw a function:

  ```
  (defun redirect-response (to-url)
    (throw 'server-response
      (lambda (stream)
        (write-redirect-response stream to-url))))

  (defmethod postprocess-response (stream (what function) headers)
    (declare (ignore headers))
    (funcall what stream))
  ```

  We can use this mechanism in `generate-response` to redirect the original request to another URL.
- Errors are passed to `handle-error` which we define thus:

  ```
  (defun handle-error (error)
    (register-error-backtrace error)
    (throw 'server-response
      error))

  (defun register-error-backtrace (error)
    (declare (ignore error))
    )

  (defmethod postprocess-response (stream (what serious-condition) headers)
    (server-error-response stream what headers))
  ```


Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Because we're in a handler-bind, the execution stack is not unwound before handle-error is called. This makes the stack available to our handler.

Exercise

My intent is for register-error-backtrace to aid server debugging by writing details of errors to a log file. Raid SWANK for code that produces error backtraces and slot this into register-error-backtrace.

Once the backtrace has been written, we throw out of handle-request; server-error-response generates a complaint and writes it to the socket stream.

Content

We'll have an easy time doing a much fancier job of this once we've looked at advanced parsing techniques (Chapter 22). For now, let's just serve a few fixed URLs, using a hash table keyed against the HTTP method and request.

```
(defvar *content* (make-hash-table :test 'equalp))

(defun find-content-handler (method request)
  (gethash (cons method request) *content*))

(defun generate-response (stream headers)
  (let* ((method (keywordify (getf headers :method)))
         (request (getf headers :url))
         (generator (find-content-handler method request)))
    (when generator
      (invoke-generator generator method request stream headers))))

(defmethod invoke-generator ((generator function) (method (eql :get))
                             request stream headers)
  (declare (ignore stream headers))
  (funcall generator request))
```

As already noted, if we wanted to serve POST requests we'd have to parse parameters from the socket stream:

```
(defmethod invoke-generator ((generator function) (method (eql :post))
                             request stream headers)
  (funcall generator request
           (parse-post-data stream headers)))
```

Exercise

Implement parse-post-data.

Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Exercise

Implement HEAD requests. Modify `write-response` so that it can send Content-Length without sending any content.

We can then write a simple macro for injecting content into the server. Even though the URL is fixed (for the moment), the response needn't be:

```
(defmacro def-content (request method form content-type)
  `(setf (gethash (cons ,method ,request) *content*)
         (lambda (,request)
           (values (funcall ,form ,request)
                   ,content-type))))

(def-content "/test" :get
  (lambda (request)
    (declare (ignore request))
    (format nil "Hello from mod_lisp on ~:(~a~)~%"
            (day-of-week)))
  :text)

(defun day-of-week ()
  (multiple-value-bind (second minute hour date month year day-of-week)
      (get-decoded-time)
    (declare (ignore second minute hour date month year))
    (svref #(monday tuesday wednesday thursday friday saturday sunday)
           day-of-week)))
```

Exercise

Write a method on `invoke-generator` for serving a URL whose response never varies (in other words, a fixed string). Write a macro akin to `def-content` for serving the contents of a file. Decide whether you'd like to derive the content type from the filename or have the site administrator explicitly declare it in this macro.

Exercise

Suppose you're serving the contents of many static files. It seems reasonable that some of these (small ones? most frequently served?) will live in memory and that others will be fetched off disk on demand. How would you organise this? Refer back to Chapter 13; are persistent objects the answer?

If we add a redirection we can test two URLs in one go:

```
(def-content "/test/redirect" :get
  (lambda (request)
    (declare (ignore request))
    (redirect-response "/test"))
  nil)
```


Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

```
ndl@vulture:~/chapter-21$ curl -L http://localhost/test/redirect
Hello from mod_lisp on Wednesday
ndl@vulture:~/chapter-21$
```

Failure Modes

We referred earlier to two functions tasked with advising the user about problems: `file-not-found-response` and `server-error-response`.

We might be tempted to define the first of these as follows:

```
(defun file-not-found-response (stream headers)
  (let* ((method (keywordify (getf headers :method)))
         (request (getf headers :url)))
    (write-response stream +http-not-found+
                    (format nil
                            "404 Not found~%=============~2%~@
                             The requested ~a URL ~a was not found on ~@
                             this server."
                            method request)
                    :text)))
```

This will certainly work but it's weak on abstraction: it can't share code with any other exception handlers and it's completely locked into producing a plain text response. Consider instead:

```
(defun file-not-found-response (stream headers)
  (let* ((content-type *default-response-content-type*)
         (method (keywordify (getf headers :method)))
         (request (getf headers :url)))
    (error-response stream content-type
                    +http-not-found+ "Not Found" headers
                    (not-found-body content-type method request))))

(defun error-response (stream content-type status title headers body)
  (write-response stream status
                  (error-message content-type status title headers body)
                  content-type))
```

These two functions will remain unchanged when we upgrade to HTML; the text generators (general-purpose wrapper for error messages `error-message` and body generator `not-found-body`) look like this:

```
(defparameter *default-response-content-type* :text)

(defmethod content-type-to-content-type ((content-type (eql :text)))
  "text/plain")

(defmethod error-message ((content-type (eql :text)) status title headers
                          body)
  (format nil "~a~2%~a~2%~a~%"
          (underline (format nil "~a ~a" status title))
          body
          (mod-lisp-identification headers)))
```


Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

```
(defmethod not-found-body ((content-type (eql :text)) method request)
  (format nil
          "The requested ~a URL ~a was not found on this server."
          method request))

ndl@vulture:~/chapter-21$ curl http://localhost/test/frob
404 Not Found
=============

The requested GET URL /test/frob was not found on this server.

SBCL 1.0.32 mod_lisp server at localhost port 80 2009-12-02 12:40:50
ndl@vulture:~/chapter-21$
```

Exercise

Implement `underline` and `mod-lisp-identification` (for which the `:host` and `:server-ip-port` headers might be useful).

Exercise

Implement `server-error-response` and an appropriate method on `server-error-body`. Inject a deliberate error into the server code and test the response. Confirm that the server is still running and use `(bt:all-threads)` to check that you're not leaking processes.

Exercise

Contemplate recoding `write-response` so that it takes a variable set of headers (you might end up using some combination of `&rest`, `&key` and `&allow-other-keys` in your parameter list). Deal specially with conversion of `content-type` and setting of `content-length`. What knock-on effect does this innocent-looking change have on the rest of the server? Which coding style do you prefer?

In the End

We've carried out our intention to implement a basic web server in about 200 lines of portable Lisp, allowing mod_lisp and Apache to take most of the responsibility for getting the HTTP right. There are some obvious gaps:

- I'll address general URL handlers and dynamic HTML generation in the next two chapters.
- For reasons of space I have not shown error handling in full and I have omitted logging altogether. We'll return to these questions in Chapter 24.

Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Exercise

What would happen if an error occurred outside the body of the `handler-bind`? (For example, the call to `handle-error` is unprotected; you might quite readily get a secondary error while trying to report the first one.) How would you consider addressing this? Your first duty is to keep the server running; recording details for posterity and informing the user are important but come in second.

If the site you're serving is small and straightforward then I recommend the mod_lisp approach.

- The code is small and all yours: easy to understand and debug.
- By working behind Apache you're presenting webmasters with familiar technology and you can leave the serving of bulk static content to them.

If you're after greater sophistication and a fully invented wheel, consider Hunchentoot.

Exercise

Instead of passing details of the request around as a property list supplemented by an input stream, implement a class for requests. Instead of passing the response around as multiple function arguments, implement a class for this as well. Which parts of the server can be simplified? Does ths code look any better for this change? What can you do with these abstractions?

Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Copyright © Nick Levine (http://lisp-book.org/) 2009 - 2011. Some rights reserved.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter
to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.