/chapter/1 In the beginning there was request? → response?

An HTTP server is responsible for accepting an HTTP request and returning an HTTP response, with some computation in between. That much you probably already knew. To put a more abstract spin on this, an HTTP server can be considered a function that takes an HTTP request as argument and whose value is an HTTP response.

That's what a *servlet* is.

Out of the box, Racket comes with two structure types, one for HTTP requests and another for HTTP responses. Using the conventional question mark, `request?` is a predicate that takes a Racket value and returns `#t` if it is an HTTP request. Similarly, `response?` is for HTTP responses. A servlet, then, if a function whose signature is

```
request? → response?
```

The Racket web server will handle a stream of bytes coming over the network and make sure that you, the programmer, get a `request?` value. Your task—should you choose to accept it—is to generate an HTTP response value.

Your job, then, is to define and combine servlets.

1.1 Servlets: big, small, and all around

Your web application, from the server point of view, can be considered as a single large servlet: a function that takes in every request whatsoever, and returns suitable responses. This suggests that servlets are 'big' functions. They carry a heavy load. As your web project grows, this one servlet gets bigger and bigger.

A more helpful perspective is to think of an HTTP server as being composed of servlets, each one devoted to handling a little part of your overall site. There's the 'main' servlet, the one through which *every* request passes. But the main servlet can *dispatch* requests to other, smaller servlets. And these servlets, in turn, can themselves be composed of other servlets.

Think of servlets the way you think of the 'main' function in a program. The main function is, of course, a function. But I'll bet that if your program has any interesting complexity to it at all, your main function will be divided into smaller functions. These smaller functions are written to help decompose our program, to make it more understandable and modular, and so on.

The same line of thinking applies to writing servlets.

1.2 HTTP requests

Requests (provided from `web-server/http/request-structs`) are structures with eight components:

Field	**Description**
`method` *bytes?*	The method (`GET`, `POST`, etc.) requested
`uri` *url?*	The requested URL.
`headers/raw` *(listof header?)*	A list of headers
`bindings/raw-promise` *(promise/c (listof binding?))*	A (promise of an) 'association list' of key-value pairs. Primarily used when processing forms.
`post-data/raw` *(or/c false/c bytes?)*	The request body. The 'post' bit is somewhat of a misnomer: a body may be present even for non-`POST` requests.

`host-ip` *string?*	The IP address of the host being requested
`host-port` *number?*	The port number of the host to which the request should be sent.
`client-ip` *string?*	The IP address of the client making the request.

1.3 HTTP responses

Responses (provided from `web-server/http/response-structs`) have six fields:

Field	**Description**
`code` *number?*	The response status code (e.g., 200, 404, etc.)
`message` *bytes?*	The 'summary' of the response. Normally goes along with the status code: if that is 200, then this will be #OK, etc. But it could be arbitrary (even empty).
`seconds` *number?*	Timestamp. The current time, in seconds, since midnight, January 1, 1970 (UTC).
`mime` *(or/c false/c bytes?)*	The MIME type for this response (e.g., `text/html` [as a sequencce of bytes.])
`headers` *(listof header?)*	Headers.
`output` *(→ output-port? any)*	The body of the response. Writes to an output port.

(For a list of standard and not-so-standard HTTP response status codes, see **the entry in Wikipedia**.)

1.4 Headers

Headers can show up in requests or responses. A header is, essentially, a key-value association, where the key and the value are byte strings.

Field	Description
`field` *bytes?*	The name of the header (e.g., `Last-Modified`)
`value` *bytes?*	The value of the header.

1.5 Conveniently generating responses

Included with this chapter is a code snippet that is used in virtually every other chapter of the book: `respond.rkt`. In that module, the main is to define a single function, `respond`, that conveniently generates Racket HTTP responses. Of course, one can always directly construct responses using `response`. But if you're like me, you may well find that approach rather nettlesome, which will cause you to want to roll your own convenience functions. (Naturally, you can do that. I make no claim on finality or superiority of my code.)

Rather than walk through `respond.rkt` here, let's be content with the fact that, in many chapters, you'll see this function called many times.

1.6 Big bites of bytes

In the discussion of requests, responses, and headers, you may have noticed that byte strings featured prominently. Why is that? Why not strings?

For instance, when extracting the method of a request, why do we get a byte string rather than, say, the string `"POST"`? That's a very simple string. Why does it have to be so byte-y?

The byte perspective makes sense because bytes are in fact what is coming to the server over the wire. Strings are, from this point of view, a non-trivial data structure, the result of 'parsing' a sequence of bytes using, say, the rules laid out in the definition of UTF-8.

Working with bytes feels 'real' and 'raw'. But it may, at times, be a bit nettlesome to constantly work in terms of bytes. One such annoyance is the conversion of bytes strings into ordinary strings. The built-in `bytes->string/utf-8` function gets used frequently. But this function doesn't (and can't!) convert *arbitrary* byte strings into strings. That is so because not every sequence of bytes is well-formed from the standpoint of UTF-8. (Continuing with the parsing idea, we know that not every sequence of characters can be parsed as a C program. Analogously, not every sequence of bytes can be understood as a UTF-8 string.)

Thus, in much of the code that you'll see in this book, there will frequently be a check whether a byte string can be converted to UTF-8 string. A function that I've found useful goes something like this:

```
(define (bytes->string b)
  (with-handlers ([exn:fail:contract? (const
#f)])
    (bytes->string/utf-8 b)))
```

(We're using `const` to make a constant function.)

The function `bytes->string` takes any Racket value as input. If it's not a byte string, then we return `#f`. If the value is a byte string, we use `bytes->string/utf-8` to get a proper string out of it; if that fails, we again return `#f`. Otherwise, we return the (converted) string.

1.7 I've written a servlet. How do I make it run?

Once you've got a servlet ready to roll, you can put it to use using **`serve/servlet`**. Here's an invocation that you'll see many times, with some variations, throughout the book:

```
(serve/servlet
   let-er-rip
   #:port 6995
   #:servlet-regexp #rx"")
```

If this function is run, you'll have an HTTP server listening for requests on port 6995 and which will call `let-er-rip` and serialize the response (that is, the value of `let-er-rip`) for you.

(The `#:servlet-regexp` bit is to ensure that *every* request received gets passed on to `let-er-rip`. The regular expression is a pattern that allows you to bypass certain patterns in the URLs. Using the empty string has the effect that nothing is filtered out.)

1.8 Servlet kata: HEAD requests

A common task for many web sites is to rewrite HTTP requests and responses. In request rewriting, one receives an HTTP request, tweaks it in some way, and passes the manipulated request on to another party. Response rewriting is similar: one receives an HTTP response, manipulates it somehow, and then passes that along to another party who is looking for a response.

With Racket, since requests and responses are structures a straightforward way to accomplish rewriting is to use **`struct-copy`**. This function takes, say, an HTTP response as input and produces a copy of it, with some details changed.

Let's see how that works in the case of `HEAD` requests.

The purpose of an HTTP `HEAD` request is, essentially, to carry out a `GET` request but return no body. Such requests are often used to determine how big a resource *would* be, if it were to be fetched with a real `GET` request.

A natural way of implementing `HEAD` is to take the request as input, rewrite its HTTP method from `HEAD` to `GET`, pass along that request, and then throw away the response body.

To pull this off in Racket, we need a few ingredients:

- a function that takes a request and changes its method from `HEAD` to `GET`
- a function that takes a response and throws away its body
- a 'core' function that works with requests in the normal way (that is, does no further rewriting shenanigans)
- a 'wrapper' function that takes a request, perhaps transforms it, and passes the transformed request to the core function.

Let's take care of these tasks one at a time.

1.8.1 HEAD to GET

This function unconditionally rewrites the HTTP method of a request into `GET`:

```
;; request? -> request?
(define (head->get req)
  (struct-copy request
               req
               [method #"GET"]))
```

1.8.2 Throw away the body

This function discards a responses body:

```
;; response? -> response?
(define (strip-body resp)
  (struct-copy response
               resp
               [output write-nothing]))
```

where `write-nothing` is the function

```
;; output-port? -> exact-nonnegative-integer?
(define (write-nothing port)
  (write-bytes #"" port))
```

`write-nothing` takes a port as input and writes the empty (byte) string to it.

The 'gotcha' here is that, for Racket responses, the body is a function. It's not simply, say, a (byte) string. That's why `write-nothing`—a function—is the value stored in the `output` field.

1.8.3 Core and wrapper functions

At this point, the responder function can be whatever you want. The mantra to keep in mind is: 'request as argument, response as value'. Let's call the core function `dispatcher`.

The wrapper function (for lack of a better word) is responsible for taking the original request as input, possibly changing some details, and passing the possibly modified request along to the core responder. Let's call the wrapper function `start`.

```
;; request? -> response?
(define (start req)
  (if (bytes=? #"HEAD" (request-method req))
      (strip-body (dispatcher (head->get req)))
      (dispatcher req)))
```

Notice that `dispatcher` gets used in either case. In case the request method is not `HEAD`, we simply invoke the dispatcher directly. If we do get a `HEAD` request, we

1. fake a `GET` request,
2. pass it along to dispatcher, and
3. throw away whatever response body comes back from dispatcher.

This is the end

You've reached the end of chapter 1 of *Server: Racket—Practical Web Development with the Racket HTTP Server*. I hope you found this a useful introduction to the Racket way of doing HTTP.

To get a copy of *Server: Racket*, you have two options:

- the *starter edition* includes 10 chapters, focusing on working entirely with Racket
- the *full-stack edition* includes all 10 chapters of the starter edition and an additional 11, covering the Racket approach to essential aspects of web development, such as relational databases and SQL, caching, sessions, Docker, and deploying to a proxy.

To purchase either one, visit **`https://gum.co/serverracket`**.