Rebuilding HTTP

By Noah Gibbs

Learn the protocol of the web by building it yourself, from the sockets to the app.

For MacOS or Linux

Ruby edition

This Version: September 2022 (free edition)

Cover Photo by Ryunosuke Kikuno on Unsplash.com

0. Why Rebuild HTTP?

Why Rebuild At All? Why This?

Here's what I said about Rebuilding Rails:

Knowing the deepest levels of any piece of software lets you master it. It's a special kind of competence you can't fake. You have to know it so well you could build it. What if you did build it? Wouldn't that be worth it?

That's still true.

Also, a lot of web programming is built on HTTP. So you're going to see a lot of it in the future. Web browsers are everywhere, and a huge number of services are provided over HTTP. That's better than you think. HTTP is a very readable, debuggable protocol, so a service provided in HTTP can be ***wonderful*** to work with. At least, it's wonderful if you know some HTTP.

You'll also need to learn to debug inside your framework and underneath. The lower levels are powerful. Learning them hands-off is really hard. By building these interfaces, you're learning as hands-on as you possibly can.

While Rack is Ruby-specific, it's also a really good interface for HTTP and application servers. Learning Rack—or a similar interface like Python's WSGI—can teach you about these interfaces.

There are a lot of historical features of HTTP and CGI. That's a polite way of saying that they were designed for situations that aren't true any more. You could do better if you designed them for the modern world. Even Rack, the youngest of the three, has some of that.

It's possible that you'd like to design alternatives to them. In that case: congratulations! Learning how to implement HTTP, CGI and even Rack can help you.

Who Should Rebuild?

I assume you enjoy the lower levels of your computer and operating system, and that you're interested in learning more. Usually that's for a job, though it doesn't have to be.

You'll have an easier time if you know a bit of Ruby. You don't need to know a ton. A lot of this will be done at the command line and use tools like curl.

There are good books to pick up Ruby if you need to, and various online classes. Some of both are free. One good starting point is https://rubyandrails.info/books if you need one.

You don't have to already know HTTP as a protocol. This will all make more sense if you've used a web browser a fair bit, and maybe some HTTP in an API. If not, fear not - I'll take you through it a little at a time.

Working Through

In early chapters we'll build parts of HTTP and Rack. Eventually, you'll have built what we call an "application server," similar to WEBrick or Puma (or many others.) Luckily building a tiny web server isn't too hard, so you'll have something useful quickly and we can expand from there.

Wherever we can, we'll look at the usual tools—curl, web browsers, frameworks and so on—and you can see how your small-but-real project can interoperate with them. That's part of how you'll learn to debug these parts of the software world.

Late in each chapter are suggested features and exercises. They're easy to skip over, and they're optional. But you'll get much more out of this book if you stop after each chapter and think about what you want to work on. Are you especially interested in one of these interfaces specifically? You can do more exercises in that chapter. Is there a specific feature that you need to work more with? You could add that as your own exercise.

Code References

Rebuilding code is great. But to understand a particular library, it helps to know the usual version of it. I'll point you at other code, documentation and so on as we go along. You don't have to read any of it to work through the book. But you're going to have a much easier time with HTTP, Rack and other code if you read other implementations, other clients, other servers, and other documentation.

At the end of each chapter will be a short section of references. You can skip them, especially on your first time through the book. But you'll get more from the book if you review that other code and documentation, at least a little.

Incompleteness

HTTP is large. There's a small, tight core that gets used constantly, and lots of obscure features that are used less. HTTP GET is easy to understand, and utterly key to "getting" how HTTP works. ETag is powerful, but you can skip it and still have a very strong grasp of HTTP.

More importantly, there are simple conceptual underpinnings, like HTTP request methods (sometimes called "HTTP verbs") and then a huge number of details that modify the basic protocol.

In this book, we're focusing very hard on the basic protocol. We will not cover every type of header field. My goal is that if you see an unfamiliar header line you'll be able to Google it and work out how to use it.

Asking "how can you implement HTTP without ETag?" is like asking, "how can you cook without cinnamon?" I promise, once you're good with a few other spices and you've eaten some cinnamon pancakes, you can figure out how cinnamon works. And once you know some other ways to do HTTP caching, ETag won't be too mind-bending.

Cheating

You can download next chapter's sample code from GitHub instead of typing chapter by chapter. You'll get a **lot** more out of the material if you type it yourself, make mistakes yourself and, yes, painstakingly debug it yourself. But if there's something you just can't get, use the sample code and move on. It'll be easier on your next time through the book.

It may take you more than one reading to get everything perfectly. Come back to code and exercises that are too much. Skip things but come back and work through them later.

There are exercises at the end of each chapter. There are answers to the exercises near the end of the book.

History

HTTP dates back to somewhere between 1989 and 1996, depending where you start counting. That means a few things have changed between then and now. There are interesting bits of background, historical accidents of design and so on.

You might want to read all about that. Or you might want to skip it and do something short-term useful with your time.

So I've put all the little historical digressions in a chapter in the back. You can read them when I refer to them in earlier chapters, or you can skip them, or you can devour them all in one sitting.

But don't worry. I've included everything you need to know in the earlier chapters. The history is to tell you ***why*** things are the way they are.

The "why" is always optional if you know the "how" well enough. And yet you can always figure out "how" if you know "why" well enough. You can decide whether you prefer one or the other or both. I'm a "both" guy, myself.

0.5 Getting Set Up

You'll need:

- Ruby
- a text editor
- a command-line or terminal
- Git (preferably)
- curl

If you don't have them, you'll need to install them. This book contains an appendix with current instructions for doing so. Or you can install from source, from your favourite package manager, from RubyGems, or Google it and follow instructions.

Nothing here uses recently-added Ruby features, so any vaguely recent Ruby is great.

By "text editor" above, I specifically mean a programmer's editor. More specifically, I mean one that uses Unix-style newlines. On Windows this means a text editor with that feature such as Notepad++, Sublime Text or TextPad. On Unix or Mac it means any editor that can create a plain text file with standard newlines such as TextEdit, Sublime Text, AquaMacs, vim or TextMate.

I assume you type at a command line. That could be Terminal, xterm, Windows "cmd" or my personal favourite, iTerm2 for Mac. The command line is likely familiar to you as a Ruby developer. This book instructs in the command line because it is the most powerful way to develop. It's worth knowing.

It's possible to skip git in favour of different version control software (Mercurial, DARCS, Subversion, Perforce...). It's highly recommended that you use some kind of version control. It should

be in your fingers so deeply that you feel wrong when you program without version control. Git is *my* favourite, but use *your* favourite. The example text will all use git and you'll need it if you grab the (optional) sample code. If you "git pull" to update your sample repo, make sure to use "-f". I'm using a slightly weird system for the chapters and I may add commits out of order.

SQLite is a simple SQL database stored in a local file on your computer. It's great for development, but please don't deploy on it. The lessons from it apply to nearly all SQL databases and adapters in one way or another, from MySQL and PostgreSQL to Oracle, VoltDB or JDBC. You'll want some recent version of SQLite 3. As I type this, the latest stable version is 3.7.11.

On a Mac you probably already have curl installed. On most Linux distributions you also will. If you don't have it, you can install on Linux with a package (e.g. Ubuntu's "sudo apt-get install curl"). If you don't have it, Google "install curl" plus the name of your OS, e.g. "install curl mac" or "install curl centos".

You may see minor differences in various tools' output, depending on version. Small differences are to be expected. Software changes frequently.

1. Listen, Connect, Read, Write

Underneath HTTP lives TCP, also called TCP/IP. In this book, we'll build HTTP from scratch on top of TCP. Ruby makes it pretty easy to build a socket server running TCP/IP, so we'll start there.

As a side effect you can make sure you have Ruby and a few tools installed and working. I'll keep this chapter simple so you're not doing too much at once.

Wait, What's TCP/IP?

TCP stands for "Transmission Control Protocol," and it's built on top of "Internet Protocol" (IP.) So that's why it's TCP/IP. That's not a very good answer, though.

TCP is a stream-based protocol. That means you write bytes on one side and they arrive on the other. But if you write 10 bytes per write, four times, it may arrive differently. It could arrive as 20 bytes each for 2 reads, or 1 read of 40 bytes or something else entirely. TCP guarantees that your bytes will arrive in the same order, but not as the same "chunks" (individual write method calls) you sent.

I'll tell you more about TCP/IP as we go along. Right now, this section is what you need to know. We can't cover every part of TCP - that's a whole book by itself to do it right! But we'll cover various things about TCP as you need to know them.

It can be a little hard to deal with TCP. But luckily HTTP isn't very demanding. We won't try to do anything too fancy where we respond to partly-arrived bytes. We'll just read until we think we have enough and then start responding.

In fact, the first thing we'll do is even easier. We'll read the request, and then we'll ignore it and send the same thing back to everybody.

A Tiny Server That's Mostly Fine

It's not hard to write a Ruby socket server. Here's a simple one that will do roughly what we want:

```
# my_server.rb
require 'socket'

RESP = <<TEXT
HTTP/1.1 200 OK
Content-Type: text/plain

Hello World!
TEXT

server = TCPServer.new 4321
loop do
  client = server.accept
  puts "Got a connection!"
  puts client.gets.inspect
  client.write RESP
  client.close
end
```

In case you don't use them constantly, that <<TEXT is called a "heredoc." Ruby will read until the word TEXT again, just after Hello World, and turn the whole thing into a string. That string is assigned to RESP.

The rest of the code opens a socket, which waits and listens for connections. Inside the loop, we accept a connection, waiting as long as necessary before one happens, then we print “Got a connection!” To the console, as well as what we received, send our response, and close the connection. Then we do it all again, with our socket, which stays open.

That’s fine, but how do we use it?

Our response above is an HTTP response, so we can use **curl**. Curl is a standard programmer and sysadmin power tool. It’s great for debugging. It’s useful in many real-world situations. By the end of this book you’re going to be pretty familiar with curl.

The server will print out what curl sends to it. You can also ask curl to print what your server sends back. That way you can see both sides. As you build more interesting servers, that’s going to be very handy.

First, run your Ruby socket server by typing “ruby my_server.rb.” It won’t usually print anything, but it will sit and wait. You’ll need to open another console window (maybe with command-N or control-N?) to run curl. Luckily, sockets exist throughout your computer so you can run servers, curl and so on in different windows without any problems.

In another console window, type “curl -v http://localhost:4321".

Where your server is running, you should see this:

```
Got a connection!
"GET / HTTP/1.1\r\n"
```

Where curl is running, you’ll see something like this:

```
*   Trying 127.0.0.1:4321...
* Connected to localhost (127.0.0.1) port 4321 (#0)
> GET / HTTP/1.1
> Host: localhost:4321
> User-Agent: curl/7.79.1
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Content-Type: text/plain
* no chunk, no close, no size. Assume close to
signal end
<
Hello World!
*Closing connection 0
```

Curl is printing out what it sent — see that "GET / HTTP/1.1" line? Your server saw that and printed it. Curl also printed out what it received, including the "Hello World!". So we have a nice start. You've written a trivial, valid HTTP server and curl can talk to it.

You've also made sure Ruby and curl are installed. You're going to need both of those constantly.

When you want to stop your server, hit control-C. That should interrupt the Ruby process and give you your prompt back.

A Little Vocabulary

I keep saying things like "what the server received" and "what the server sent back" and so on. That's going to get annoying.

The official HTTP names for these things are Request and Response. A Request is what the client sends to the server, and the Response is what's sent back.

HTTP is mostly one-way. You don't have a response ***to*** the Response. If the client gets the Response and wants to do something because of it, it sends another Request.

HTTP is Weirdly Readable

Here's something unusual about HTTP: it's not hard to read and write it as a human. You'll occasionally see operations engineers just open up a socket to a web server and request things by typing HTTP lines at their keyboard. Or you'll see little chunks of HTTP in a bash script, just being sent around to one place or another.

Relatedly, curl has the "-v" (verbose) option to just print everything to the console. That's not a thing you can do with every protocol.

A lot of protocols are full of non-human-readable binary data structures and byte values that don't mean anything if you print them out. HTTP is basically readable and basically writable.

What's Still Wrong?

I mentioned that not everything is right here. What's not working? Several things.

For instance, notice how curl printed out several more lines than our server did? That's because curl sent several lines (GET, Host, User-Agent, Accept) but we used "client.gets" to read, so we only got one. Gets will read one line (just until it sees a newline) and then stop.

I mentioned that TCP/IP makes some things difficult. Remember that with TCP we don't know how many bytes are coming, or how they were written. We just get a stream of them in order. Our server will read until it sees the first newline—one line. But we'll need to know a little more about HTTP to correctly get the whole thing.

It would also be nice for us to pay attention to what the request said, not just always return the same thing. We'll get there.

Curl prints "no chunk, no close, no size. Assume close to signal end." There are several ways you can tell HTTP how long your response is. You can tell HTTP a size in bytes. You can tell it to use "chunked" encoding. And you can close the connection. We close the connection for this server, and then curl knows we're done. But before we do, it complains that it can't be certain when we're done yet. Setting a size would be more polite.

It also prints "mark bundle as not supporting multiuser." That's fine. In this book we're implementing HTTP/1.1 instead of HTTP/2.0, so we don't support multiplexing. Multiplexing is a way to request and return multiple files over the same connection. And for this book we won't support that, so that message is fine.

Why won't we get into HTTP/2.0? Well, remember when I said that HTTP is weirdly readable and writable? HTTP/2.0 is ***not***.

I am partly in the business of teaching you mental models, and fingertip feel for what the computer is doing. I am trying to get you comfortable sticking your eyes and typing-fingers into places most people don't look or poke. HTTP/1.1 and older are ***fantastic*** protocols for that. HTTP/2.0 is not.

For the same reason we won't be implementing HTTPS, which is the secure encoded version of HTTP.

What's Fine But Weird?

Some things aren't wrong exactly, but might still surprise you. For instance, did you notice the newlines there? Usually we write a newline as \n (backslash-N) and "inspect" prints it out the same way. But here we're seeing \r\n (backslash-R-backslash-N) from Ruby, but not from curl. Why?

It turns out HTTP newlines are always written that way. If you want to know why, have a look at the "Historical Accidents, Background and Stories" chapter at the end. For now, I'll just say that's how HTTP does it.

HTTP clients send them that way, and so should HTTP servers.

That means we'll need to send that style of newline for a correct HTTP implementation. Right now we aren't. So this is ***both*** something weird-but-fine and something wrong, all in one go! But I had to explain the weird one before the wrong one made any sense.

Curl already knows those are normal HTTP newlines, so it doesn't bother to print them out specially. But in Ruby that's a weird way to do it, so inspect makes sure you can see they're weird.

That also means you have a way to tell if you're doing it right: inspect should show the special HTTP newline, if you've composed it correctly.

Right now curl is being error-tolerant. That means it's letting you get away with the wrong newlines, for now. But we should fix the problem, because not every HTTP client is as forgiving as curl.

What Does That GET Line Mean?

HTTP has a lot of pieces. But the first, simplest thing is that GET line. It looks like this: “GET / HTTP/1.1”. There are three interesting pieces there. The first is GET.

GET is an HTTP request method, sometimes called an HTTP verb. Verbs in English are action words like “run” or “speaks” or “wiggled.” HTTP has a few different verbs. The simplest is GET, which requests a file from a server. The first line starts with the verb.

In addition to GET, there are other common ones like PUT and POST, and less-common ones like DELETE and PATCH and HEAD.

You could say “head isn’t really a verb, or even a method,” and I’d agree with you. But in HTTP we pretend it’s one. HTTP and English don’t always agree about everything, such as the spelling of “REFERER.”

The second important part of that line is the slash between GET and HTTP. If you request a different URL from curl, you’ll get something different there. Try it! Run your server again if you’ve shut it down, and then try “curl http://localhost:4321/some_url". Your server will print a slightly different line: “GET /some_url HTTP/1.1\r\n”.

So the slash isn’t a separator between GET and HTTP in that line. It’s the path you’re asking the server to send. If you don’t ask for any specific path, you get / (slash), which is often called the “root” path.

The final interesting thing in the line is, of course, HTTP/1.1. If the number after HTTP were different, your server would have to treat it differently. Really old programs might send HTTP/1.0 or

HTTP/0.9. But HTTP/1.1 has been around for a very long time now, and basically every server accepts it.

As I mentioned earlier, HTTP/2.0 is a thing now. It's much harder to write a server for it, and we won't in this book. But it's built on HTTP/1.1, and you'll understand it much better after you learn. We'll talk about some of those differences later.

Exercises

Exercise One: HTTP Request Methods

Curl is ridiculously customisable. You can specify your own verb ("request method") for instance. Try running your server, but then for curl, do "curl -X DISCO http://localhost:4321/FEVER".

Instead of printing GET / HTTP/1.1, it should print "DISCO / FEVER HTTP/1.1". While HTTP servers have specific methods they respond to, nothing stops you from sending anything you like. The server is likely to tell you it's never heard of it, though.

You can check. You can point curl at https://google.com with the verb "DISCO_FEVER" and see what they say. Give it a try.

However, there are limits. If you try to use "DISCO FEVER" with a space as the verb, you'll get a protocol error. Give that a try, too. For me, at least, curl will cheerfully attempt it and Google will tell me "no chance."

Exercise Two: Networking

I claimed these are network sockets. But you're only messing with them on your own computer. How do you make it work somewhere else?

You've already done part of the work for that, it turns out. It works fine with the right configuration already. But there's still more to do if you want to test it.

You'll need two machines that can 'see' each other on the network, and it's easiest if at least one has a DNS name, something like "my machine.whatevernetwork.com".

Once you have that, you'll run the server on one machine and curl on the other. Then instead of writing "localhost" you'll write the name of the machine with the server running, something like "curl http://mymachine.my_isp.net:4321". Notice that you still need the 4321. That's the port number **on** that machine, which is how we tell different network servers apart when they run on the same computer.

Don't be too discouraged if it doesn't work. You need the DNS name, and the port needs to be open. Sometimes there's a firewall running, which blocks unexpected connections on unexpected ports. That's a way to keep random servers, like the one you wrote, from running when they might be insecure.

Your server is ***extremely*** secure right now, since it does almost nothing. There's no way to abuse it. But your firewall doesn't know that.

If you can configure your firewall, try unblocking port 4321. It's also possible to run your server on a port that would usually be unblocked. But it's hard to find an unblocked port that another service isn't using already. Often you only unblock a port for a specific server you know will be using it.

Code References

You might ask, "who does this for real?" For instance, does Ruby on Rails parse HTTP? As it turns out, no, it does not. Instead,

frameworks like Rails use libraries like Rack for even simple things like HTTP parameter parsing. But even Rack doesn't normally parse the HTTP request that you're seeing here.

Instead, you have a program called an Application Server that reads HTTP and then passes requests to Rack, and then to your framework and application. Right now our app server, our interface, our framework and our application are all mixed together in a single file. Over time we'll start to separate out the pieces and it will be clearer what each one does.

The simplest Application Server you'll normally see in Ruby is called Webrick (or WEBrick.) Webrick used to be part of Ruby, but now it's part of a separate gem. It's still commonly used for simple situations where you don't need a lot of performance.

Here's the source code: https://github.com/ruby/webrick

If you want to see HTTP handling done "for real," Webrick is a pretty good place to start looking. It does more than our application server in this book will. For instance, it handles HTTP proxying and HTTPS. But it's a fairly simple application server that doesn't handle concurrency. Unlike Webrick, Puma uses multiple processes and multiple threads per process. Falcon uses Fibers for similar reasons, Thin uses EventMachine, Unicorn uses worker processes... In all these cases, the concurrency is a huge source of complexity which Webrick avoids.

I wouldn't use Webrick for production. But it's an excellent place to find code for how an application server works.

For instance, you can see the HTTP status codes and whether each one is a success, a warning, an error, a redirect and so on. Just check https://github.com/ruby/webrick/blob/master/lib/webrick/httpstatus.rb and you'll see the code.

Or you can find out the various attributes of a cookie in how it parses cookies: https://github.com/ruby/webrick/blob/master/lib/webrick/cookie.rb.

Webrick's code is, intentionally, far more advanced than the rest of this chapter. I'm not saying "go and read all of Webrick." But it's a great starting point for understanding the real thing.

2. Reading a Request

In the last chapter, curl was our HTTP client. We'll use it for that a lot. It's not hard to write a client, either. But curl is so flexible, so standard, so available and so useful that we'll be using it a lot for basically the whole book.

Using curl proves that your HTTP server can talk to real, normal HTTP clients. That's a good thing.

Let's talk a little more about the Request that curl sends to our server and how to read it.

Sample Source

Sample source for all chapters is on GitHub.

https://github.com/noahgibbs/rebuilding_http

Once you've cloned the repository, do "git checkout -b chapter_2_mine chapter_2" to create a new branch called "chapter_2_mine" for your commits.

If you already have last chapter's code you don't need to do this. But every chapter from here on will have a link to source code if you want to start there.

TCP is Not Our Friend Here

Currently we're using "client.gets" to read until we get a newline. Can't we just read the whole thing and then stop?

Not without knowing a little more, no.

TCP/IP will keep trying to deliver bytes in a stream, but you can't know whether more is coming. TCP doesn't know either. I mean,

maybe you're just about to write more to that socket. How would it tell? For that matter, if half the request has arrived, the other half may take microseconds, or seconds, to arrive. TCP doesn't know what else is on the way (or not.)

Instead, in stream-oriented protocols like TCP, you need lengths or separators. If you write a length before your message, that tells the receiver how much to read.

Or if you have a separator, the receiver can just keep reading until it sees the separator. For instance, if your separator is "bbbbbbbb" then you could read until you saw that. But that only works if there's no "bbbbbbbb" in your message, so you have to pick your separator carefully.

HTTP's client-to-server protocol uses a separator: two newlines in a row. If you're printing to console then a blank line marks the end of the request. So now you know: the HTTP request can't ever contain two newlines in a row, because that would mean end-of-request.

One way to look for two newlines is to read line-by-line until you get an empty line. So let's do that:

```
# my_server.rb
require 'socket'

RESP = <<TEXT
HTTP/1.1 200 OK
Content-Type: text/plain

Hello World!
TEXT

server = TCPServer.new 4321
```

```
loop do
  client = server.accept
  loop do
    line = client.gets
    puts line
    break if line.strip == ""
  end
  client.write RESP
  client.close
end
```

"Strip" is just a Ruby String method to remove whitespace from the start and end of the string, so it'll get rid of the \n\r for us.

If you run this again and run curl, your server should print out the whole request including the blank line at the end. So you should see the Accepts line too, for instance, not just the GET line by itself.

You can also run curl multiple times, either with this server or the chapter 1 server, and see that it works repeatedly. Good. Servers are supposed to do that.

Libraries and Repeat Code

Is this code starting to get long? I feel like this code is starting to get long. We need a library file to put our constants and methods in. That may save you some retyping or careful reading.

Ruby likes to put files like that under "lib". That's where you'll see them in gems (Ruby libraries.) You should make a directory for your Rebuilding HTTP code if you haven't already, and then make a "lib" directory under that.

Now let's pull out a little code into a new file, which we'll call lib/rebuilding_http.rb:

```
# lib/rebuilding_http.rb
require "socket"

HELLO_WORLD_RESPONSE = <<TEXT
HTTP/1.1 200 OK
Content-Type: text/plain

Hello World!
TEXT

module RHTTP
  def read_request(sock)
    out = ""
    loop do
      line = sock.gets
      out << line.chomp << "\n"
      return(out) if line.strip == ""
    end
  end
end
```

Chomp, above, removes newlines at the end. So we're removing the \r\n or other newline and adding our own Mac-style newline.

That's fine. Now let's use it for our simpler server:

```
# my_server.rb
require_relative 'lib/rebuilding_http'
include RHHTP
```

```
server = TCPServer.new 4321
loop do
  client = server.accept
  req = RHTTP.read_request(client)
  puts req
  client.write HELLO_WORLD_RESPONSE
  client.close
end
```

I find this code a lot cleaner and simpler to understand. And by writing methods as we go, you can skip a lot of repeat typing. Debugging is useful, and it's good for you. But retyping the exact same thing over and over isn't as good.

You ***are*** typing this out and debugging it, ***right?*** That's the part that improves your skill.

We're using require_relative above to avoid messing with paths. The require_relative looks for the file at a path relative to the current source file, not the current directory or your gems or your include path. That way you can find exactly the file you want, if it's in the same project or gem. Require_relative is a good trick to know. We'll use normal requires later when we build a gem, since require_relative doesn't work as well for that.

We're also using "include RHTTP" there to include methods from our module into the top-level namespace. That's slightly unusual, but it works well in an ebook where our horizontal space is quite limited. We'll fix that too when we build a gem.

We gave a better name to HELLO_WORLD_RESPONSE. It used to be RESP. When you pull something out to reuse, that's a great time to name it. Small things used locally can have small,

unobtrusive names. But if you'll have to look through the code for something, you want it to have a memorable, descriptive name.

We're still leaving it as a top-level constant. That's because it's a "hello, world" response—it's not going to be part of the long-term API. It's going to go away completely, very soon. It's fine that it looks out of place. That makes it even more likely that you'll want to get rid of it.

It's not a bad thing when messes look like messes.

It's also not bad to make messes while you work as long as you clean them up afterward.

We also gave read_request a name so it's clearer what we're doing and why.

So far we're not getting anything useful ***out*** of the request. That comes next.

Request Parsing

Let's assume that neither the request method nor the URL will have whitespace. This is true, basically always.

So then we can cut up our request line where it hits whitespace. We've already covered some useful Ruby string processing. Below, we'll use another fun one.

So: let's make a Request object and have it parse our request. That changes the top-level server, but not much:

```
# my_server.rb
require_relative 'lib/rebuilding_http'
include RHTTP
```

```
server = TCPServer.new 4321
loop do
  client = server.accept
  req = RHTTP.get_request(client)
  puts req.inspect
  client.write HELLO_WORLD_RESPONSE
  client.close
end
```

When I write a chunk of code and mark it “excerpt,” like below, you’ll want to add it to your library file. You can replace old methods if there’s one with the same name. It will basically work if you just keep adding more methods - Ruby will cheerfully replace an old method if you define a new one with the same name. When in doubt, you can check the Git repo for the following chapter.

But I write “excerpts” to let you know that other methods with different names should stick around. We’ll use our read_request method from earlier this chapter here, so don’t delete it.

```
# lib/rebuilding_http.rb (excerpt)
module RHTTP
 def get_request(sock)
    req_text = read_request(sock)
    RHTTP::Request.new(req_text)
  end
end

class RHTTP::Request
  attr_reader :method
  attr_reader :url
  attr_reader :http_version
```

```
    attr_reader :headers

    def initialize(text)
      lines = text.split("\n").flat_map { |s|
        s.split(“\r")
      }
      @method, @url, rest = lines[0].split(/\s/, 3)
      if rest =~ /HTTP\/(\d+)\.(\d+)/
        @http_version = "#{$1}.#{$2}"
      end
      @headers = lines[1..-1].join("\n")
    end
  end
```

There are a few interesting things happening here. The get_request method takes a socket, reads the text and then makes a “Request” object. That seems simple enough.

The Request object declares some properties, then parses them from the text in initialize. Fair.

There’s a somewhat complicated double-split. I’ll tell you two things about it. First, it’s just splitting on any combination of \n, \r or \n\r, while keeping the empty line in between for doubles. Second, you should ***never trust me on anything like that without checking for yourself***. In a minute I’ll show you how to check for yourself.

We’re also doing a split on whitespace which is also worth checking for yourself. You’re learning about Ruby infrastructure, and as far as I’m concerned that means you’re going to finish the book with a lot of interesting string-parsing techniques in your head. If we were learning C it would be bitwise operations, but in Ruby it’s string parsing. You should know the things your language is good for.

Finally we check if there's an HTTP/1.1 or similar in the final bit of the request to check the HTTP version, and save all the lines after the first one as headers. Those parts are simpler.

Why You Should Never Trust Me

I could tell you that I'm just some guy, not a real authority, and that's probably true. But there's a far more important reason not to trust me: I'm writing from the past. Ruby changes. Heck, even HTTP changes over time. By nature, you can't be sure I'm giving you good advice because even good advice goes bad over time.

The first line of an HTTP request is, admittedly, unlikely to change. But the ***habit of distrust*** is the important part. Give a person a bit of distrust and they'll double-check a line of text, but ***teach them to distrust*** and they'll double-check for a lifetime.

So: whenever you read technical information, assume the person talking is your dodgy ex-boyfriend, using a whiny, nasal tone of voice. Yes, ***even if you don't date men***. That's how much you should trust random technical advice.

How You Should Avoid Trusting Me

It's easy for ***me*** to say not to trust me. I'm the one with a vast treasure-hoard of Ruby knowledge. But how do ***you*** check?

The answer is irb.

When you installed Ruby, you probably installed a program called irb with it. If you type "irb" and it's not installed, Google "install ruby irb MyOperatingSystem" and then fix the problem. It's cool, I'll wait here.

Here I am, with my nasal whine, telling you that that double-split code does what I say it does. Instead of listening, open up irb:

```
> irb
irb(main):001:0> def test_lines(s);
s.split("\n").flat_map { |s| s.split("\r") }
; end
=> :test_lines
irb(main):002:0> test_lines("a\n   b  \r  c  \n  d
\n\r\n\r e")
=> ["a", "   b  ", "  c  ", "  d ", "", " e"]
```

So: I wrote a method called test_lines that takes a string and then does that double-split thing on it. Then, I called it on a big string with various newlines in it to see what it did. And lo and behold, it split it up just like I said, including the part where two consecutive \n\r chunks left an empty line in between.

And you ***actually typed it in and checked***, right? Otherwise you're being charmed again by my nasal whine. That's a bad idea with your dodgy ex-boyfriend. I'm just going to ask you for another loan at this rate, and I never paid back the first one.

I also said some things about the line that goes "@method, @url, rest = lines[0].split(/\s/, 3)". This would be a great time to define a method that does that split and play with it in irb. Don't let me just say things, make me work for it.

If you shouldn't trust me, does that mean I ***don't*** really have a vast treasure-hoard of Ruby knowledge? Possibly. Hey, this is unrelated, but did you know that if you're writing a book you can just type things into Ruby, see what they do and then pretend you knew the whole time?

If somebody tells you cool Ruby facts, step one is go check in irb.

Wrapping Up

I think this chapter is long enough. It's nice to keep projects small, isn't it? You can always start on another chapter, but breaking in the middle is awkward.

You've learned a bit about how to parse a multi-line request using TCP sockets. You've broken off reusable code into a library file, and put your request into an object. You might have noticed ***why*** your request methods and URL can't have spaces in them, and you can play around to see how that would (not) work.

More importantly than any of this, you've learned good reasons to distrust people writing about software. And some irb, but the distrust is the important part.

There's a lot more to HTTP requests. But next chapter we'll learn a bit about responses.

Exercises

Exercise One: Parameters

We haven't talked about HTTP URL parameters. Sometimes you'll type, say, "https://my_site.com?search=roast_penguin" into the browser. The part after the question mark is the HTTP parameter(s).

First let's see how curl handles them. Run your server and type this:

```
curl http://localhost:4321?search=delicious
```

There's a decent chance you'll get a message like "zsh: no matches found". Doesn't look like you're touching your server, does it? That's because you aren't.

Now try it with double-quotes around the URL:

```
curl "http://localhost:4321/?search=delicious"
```

Computers are tricky devils. Now, what does your server see as the URL?

Exercise Two: More Parameters

Curl can do more tricks with parameters. Normally if you set parameters with "-d" it will send a POST request, which is a little different from what we're doing. But you can tell it not to with "—get" (that's a double-dash, or dash-dash-get.)

So try pointing curl at your server with this command line:

```
curl ––get –d search=delicious “http://
localhost:4321/my/url?page=7"
```

What does it do to the URL? What happens with the parameters?

If you're feeling adventurous, get rid of the --get and let it send a POST.

Exercise Three: Basic Authentication

HTTP has a bad built-in authentication method, just called "basic" authentication. You should never use it. Curl has it built in, of course. Curl does everything, not just good ideas.

Run your server and point curl at it:

```
curl -u bobo:bobopwd --basic http://localhost:4321
```

Look through the result, especially the headers. What do you see?

It may look sort of secure. There's no "bobopwd" that you can read, right? But it's just a base64 encoding of the username and password.

So please, please don't think that HTTP's basic authentication is even slightly secure.

Code References

Once again Webrick is an interesting read. If you check its httprequest.rb (https://github.com/ruby/webrick/blob/master/lib/webrick/httprequest.rb) and search for read_request_line, you can see how it handles the first line of an HTTP request.

It uses a regular expression to parse the entire line, checking for whitespace in between the method and the URL. Nearly everything else is optional up until the final newline, including the carriage return before it. You can also look for "read_header" to see how it handles the equivalent of that split you did.

In its httputils.rb, you can look for parse_header and see how it reads the header lines after the request line. Once again, it uses regular expressions.

Ruby regular expressions are quite fast. In most high-level languages, regular expressions run a lot faster than native code so you'll see them used extensively for this kind of parsing.

In lower-level languages like C the speed advantage disappears. But parsing code in C is hard to write so you'll still see regular

expressions and parser generators used pretty frequently. It’s just easier to use a specialised tool where you can.

We've talked about Webrick a few times. Aren't there other application servers? There definitely are. For instance, Puma is probably the default to use for most new projects, especially if you want good performance. Can we look through Puma?

Puma's code is also online: https://github.com/puma/puma. And you can look through it. But because Puma is meant for real-world production use, it often involves much more complicated approaches. You can read https://github.com/puma/puma/blob/master/lib/puma/request.rb, but it doesn't contain much actual HTTP parsing. Instead, Puma uses a complex parser-builder to get much faster results. You can look through it in https://github.com/puma/puma/tree/master/ext/puma_http11, but it will be a lot harder to answer questions like "how does Puma separate out the GET from the URL and HTTP/1.1?"

If you read closely through files in that directory, you'll see a copyright from Zed Shaw, author of "Learn Ruby the Hard Way" and an old-time Rubyist from way back. Puma took the HTTP parser from Mongrel, a much older app server. That's part of a pattern: in open source we re-use and adapt older pieces of code that worked well.

But all of this combines to make Puma a lot harder to read if you don't already understand what it's doing. It's the same reason that we didn't start this book with "let's read through the code to a production app server!"

3. A Quick Response

HTTP requests come ***to*** your server. Responses get sent back to the client. We have an idea of what some requests look like. We've only looked at that one lone response so far.

I also said we'd get rid of that "Hello, World" response before long. The time has come. It's bothering me to see it in the source code. That's good! I intentionally made it look out-of-place. So it kept bothering me. And soon we'll fix it.

But first, what do responses look like?

Sample Source

Sample source for all chapters is on GitHub.

https://github.com/noahgibbs/rebuilding_http

Once you've cloned the repository, do "git checkout -b chapter_3_mine chapter_3" to create a new branch called "chapter_3_mine" for your commits.

What's In a Response?

We have our little server to see what requests look like. How can we see responses?

Hey, remember how curl keeps printing out that "Hello, World?" If we use "curl -v" it will print out a lot more.

While you could use curl on various public servers and see what they return, that's going to be a lot of output. Web pages are often quite large. You want the smallest page you can get.

So we'll use Webrick. It used to be Ruby's built-in web server. It's pretty straightforward.

Make a new file called webrick_local.rb:

```
# webrick_local.rb
require 'webrick'

root = File.expand_path(".")
server = WEBrick::HTTPServer.new Port: 4322,
  DocumentRoot: root
trap('INT') { server.shutdown }
server.start
```

If you run this with "ruby webrick_local.rb" it will start serving the files from your current directory on port 4322. But first you'll want a file to serve.

Here's a nice trivial one, which we'll put in index.html:

```
<!-- index.html -->
Hello in <b>HTML</b>!
```

Now if you run your server, you can grab it with curl:

```
curl -v http://localhost:4322
```

Curl's output should end with "Hello in HTML!". That's what you wanted. Let's have a look at what's ***before*** that, though.

Here's what I see:

```
$ curl -v localhost:4322
*   Trying 127.0.0.1:4322...
* Connected to localhost (127.0.0.1) port 4322 (#0)
> GET / HTTP/1.1
> Host: localhost:4322
> User-Agent: curl/7.79.1
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Etag: d6e574-2a-632e3e21
< Content-Type: text/html
< Content-Length: 42
< Last-Modified: Fri, 23 Sep 2022 23:15:45 GMT
< Server: WEBrick/1.7.0 (Ruby/3.0.2/2021-07-07)
< Date: Fri, 23 Sep 2022 23:15:50 GMT
< Connection: Keep-Alive
<
<!-- index.html -->
Hello in <b>HTML</b>!
*Connection #0 to host localhost left intact
```

The lines with “>” are what curl sent to the server. They should look like the requests you’ve seen.

The lines with “<“ are the response that the server sends back. The first line of a response is special, just like a request. Instead of “GET / HTTP/1.1”, though, it says “HTTP/1.1 200 OK”.

The HTTP/1.1 is the same, obviously. 200 is the HTTP status code. A 200 means “no error, everything went fine.” And the OK

afterward is just an explanation of what the status means. The explanation is written for people. Your code is just going to check the status number. But you might display the words to a person, since “200” and “404” and “500” are harder for people to interpret.

Then there are all those lines with a word, a colon, and more stuff afterward. You may remember that for our Request object we read those, tucked them away into a variable and didn’t do anything with them… yet.

And then there’s a blank line, just like for the request. But for the response, there’s more ***after*** the blank line. That’s where our tiny little web page shows up. It had to be in there somewhere!

You’ll notice that curl doesn’t print an extra “<“ in front of the web page part. In fact, if you get rid of the “-v” curl will still print those. That’s because curl assumes that’s what you actually wanted. All these headers are what you want right now, but a normal user would probably prefer to just see the web page.

You can change what’s in index.html and run curl again, just to see how things change. A lot of the point of rebuilding HTTP is to get a feel for how this works. And now you can!

You shouldn’t need to restart the Webrick server. If you change the Ruby code, you’ll need to restart a server. But it’s reading the HTML files every time, so no restart is needed.

A Less-Canned Response

This was fun, but what about our other server? Have we abandoned it for Webrick?

Not at all.

We'll hop back to it and put together a quick Response object. Our Request object took a chunk of HTTP text and parsed data out of it. Our Response will do the opposite. We'll initialise it with data fields and it will give us the text to send back.

Keep in mind that for these excerpt files you're not deleting all the old code, just modifying or replacing.

```
# lib/rebuilding_http.rb (excerpt)
class RHTTP::Response
  def initialize(body,
      version: "1.1",
      status: 200,
      message: "OK",
      headers: {})
    @version = version
    @status = status
    @message = message
    @headers = headers
    @body = body
  end

  def to_s
    <<RESPONSE
HTTP/#{@version} #{@status} #{@message}
Content-Type: text/html
#{@headers.map {|k,v| "#{k}: #{v}" }.join("\n\r")}

#{@body}
RESPONSE
  end
end
```

And now that we have a response object, let's use it:

```
# my_server.rb
require_relative 'lib/rebuilding_http'
include RHTTP

server = TCPServer.new 4321
loop do
  client = server.accept
  req = RHTTP.get_request(client)
  puts req.inspect
  resp = RHTTP::Response.new("Hello from Ruby!",
    headers: { 'Framework': 'UltraCool 0.1' })
  client.write resp.to_s
  client.close
end
```

Go ahead and run your server: "ruby my_server.rb". When it's running, use "curl" again: "curl -v http://localhost:4321". Now you should see "Hello from Ruby!" in curl's output.

Heading Out

Both requests and responses have a lot of lines after the first line, which start with a word (or a few) and a colon. The whole chunk is called the header. You'll also see the lines called 'headers'.

For requests, we acted like the header lines are just a bunch of lines of text. Which is true, kind of. Here's one set, for instance:

```
Host: localhost:4321
User-Agent: curl/7.79.1
```

```
Accept: */*
```

Yup, those are clearly lines of text. But each one means something. 'Host' is different from 'User-Agent' is different from 'Accept.' You'll see each one at most once (usually.)

So headers are less like a bunch of lines and more like a hash table. We have the type of each header field (key) and the value for that header field (value.) That's a better way for us to do it.

For responses, and soon for requests, we'll use a hash table. And then we can override header fields, as we do above with 'framework.' You're not required to name yours 'UltraCool.' And the opposite of 'required' is a ***luxury***, right?

Standards and Living Languages

We're talking a lot about HTTP, a widely-used open standard, and we haven't mentioned the "standard" part much. There is a standards document. We haven't looked at it yet. ***And we won't***.

I also keep mentioning things with more than one name. Is it a "request method", or an "http verb"? Is the whole block of lines a header, or is there an 'Accepts header'?

Several things are happening here. The standards document is absolutely unreadable. It's not designed for random people to read. It's terrible for that. Please don't bother.

But there's something even more pernicious: HTTP is a living language. Nobody stops anyone from releasing an incorrect implementation. It's easy to ***write*** one, but hard to understand the official rules. You don't care much if your code fits the standard. You care if your code works with all those other mostly-compatible implementations.

That means the standard is... optional. Secondary.

A living language has lots of synonyms and repetition. A pedant could easily say "there's an official name, so stop using the other ones!" But in a living language if you don't know the other ones, you're not "more correct." You're "partially not fluent." It's like if you truly didn't recognise "gonna" as a synonym for "going to" in spoken English. That's not a superpower, it's a weakness.

So if the standard is hard to read, is that a problem? Not really. There are lots of good implementations. You can already read some of them. Where common implementations disagree with the standard, you should follow the implementation. If the standard says some particular format is allowed and Apache won't handle it, it's not useful to you. If the standard says some particular construct isn't allowed and curl routinely generates it, you should handle curl's output.

In a few cases you might **dis-**allow something uncommon that the standard forbids. But the way you'd check for "uncommon" is to see which ***implementations*** might produce it. ***The standard is a description, not the authority***.

That's not always true. But it's true for living languages.

Exercises

Exercise One: Status Codes

Remember your local-webrick server? What if you sent it a different request method?

Run the server and then point curl at it:

```
curl -v -X BAD_METHOD localhost:4322
```

Find the first line of the response. Notice that you have a status number and then a human-readable explanation of that status. Can you find other curl command lines that give other errors?

Exercise Two: Overriding Headers

We didn't really override any headers. We only sent headers if the server directly specified them, like it did with 'framework.' How would we make headers overridable?

Try making a 'Date' header that gets set automatically in the framework, but if you set it in my_server.rb, the server will win. You can require "time" and use Time.now.httpdate to get an HTTP-formatted date, because Ruby is awesome.

You'll want to use a recognisably different date to see if it's being set properly. You can subtract a number of seconds from a Ruby date, something like this:

```
# give me a date from last year
d = Time.now - (60 * 60 * 24 * 365)
```

Exercise Three: the Browser

It's a bit odd using HTTP so much from the command line, isn't it? This method is ridiculously good for debugging. But of course, the browser is the point of all of this.

Have you already tried pointing the browser at your server? You can do that, you know.

If you point the browser at your server, you'll see a cheerful "Hello from Ruby!", just like you did from curl. But of course, that's not the most interesting part.

Have a look at the console where your server is running. The browser's HTTP request is ***huge***. Also it sends more than one, since it looks for favicon.ico, not just the URL you gave it.

Here's what one of my two looked like:

```
#<RHTTP::Request:0x00007fc0f40c1b30 @method="GET",
@url="/", @http_version="1.1", @headers=["Host:
localhost:4321", "Connection: keep-alive", "sec-ch-
ua: \"Google Chrome\";v=\"105\",
\"Not)A;Brand\";v=\"8\", \"Chromium\";v=\"105\"",
"sec-ch-ua-mobile: ?0", "sec-ch-ua-platform:
\"macOS\"", "DNT: 1", "Upgrade-Insecure-Requests:
1", "User-Agent: Mozilla/5.0 (Macintosh; Intel Mac
OS X 10_15_7) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/105.0.0.0 Safari/537.36", "Accept:
text/html,application/xhtml+xml,application/
xml;q=0.9,image/avif,image/webp,image/apng,*/
*;q=0.8,application/signed-exchange;v=b3;q=0.9",
"Sec-Fetch-Site: none", "Sec-Fetch-Mode: navigate",
"Sec-Fetch-User: ?1", "Sec-Fetch-Dest: document",
"Accept-Encoding: gzip, deflate, br", "Accept-
Language: en-US,en;q=0.9", "Cookie:
_ga=GA1.1.830734641.1662617561"]>
```

We'll look at some of those headers later. But keep in mind: if you run an HTTP server, you get to see everything that's being sent back and forth.

For the exercise, point your browser at your http server and copy and paste the results. Have a look at all the header lines you get, and see how they differ from mine.

Code References

I keep suggesting that you read Ruby application servers. That's a good idea, but it's not the whole story. Curl is ***also*** open-source. You can find it here: https://github.com/curl/curl

Curl is written in C, which may make it harder for you to read. It certainly means there are a ***lot*** of lines of source code for many operations. But you can still find some interesting things.

For instance, what does Curl know about cookies? You can check its cookie header file, https://github.com/curl/curl/blob/master/lib/cookie.h, and look at the structure at the top. Hey look, there are all the data fields for a cookie.

And then there are ALL_CAPS constants with comments about them. MAX_COOKIE_LINE is documented pretty well, for instance. If you don't know HTTP cookies it may not make much sense yet. But hang on, we'll talk about them a bit later.

In the mean time, curl has interesting stuff to look through. If you find C code hard to read, look for filenames ending in .h, which are called header files. "Header" is unrelated to the headers in the requests and responses. Instead, C headers have a list of data structures, constants and what functions (methods) exist to be used.

So a header can make a good summary of what's available in a C codebase, even if you're not ready to read thousands of lines of code to find out exactly how each function works internally.

4. POST Requests

We can parse requests. We can return responses. What more is there to HTTP?

Lots of things, naturally.

GET requests will get you pretty far. Some things can be done with ***only*** GET requests. But before long you'll want to send more data back to the server.

As you may know, that kind of thing is done with POST requests, and their relatives like PUT, PATCH and DELETE.

There are lots of reasons to use requests other than GET. Let's get started on both how and why.

Sample Source

Sample source for all chapters is on GitHub.

https://github.com/noahgibbs/rebuilding_http

Once you've cloned the repository, do "git checkout -b chapter_4_mine chapter_4" to create a new branch called "chapter_4_mine" for your commits.

What Do POSTs Look Like?

"Curl -v" doesn't show you everything it sends to the HTTP server, but that doesn't mean curl can't do that. Let's learn a little more about curl and about POST requests.

First, run your server with "ruby my_server.rb" like you've been doing. Then run curl: "curl --trace-ascii logfile1.txt http://

localhost:4321". You should see "Hello from Ruby!" from your server.

And now that logfile, called logfile1.txt, has interesting things in it, something like this:

```
== Info:   Trying 127.0.0.1:4321...
== Info: Connected to localhost (127.0.0.1) port
4321 (#0)
=> Send header, 81 bytes (0x51)
0000: GET /url HTTP/1.1
0013: Host: localhost:4321
0029: User-Agent: curl/7.79.1
0042: Accept: */*
004f:
== Info: Mark bundle as not supporting multiuse
<= Recv header, 16 bytes (0x10)
0000: HTTP/1.1 200 OK.
<= Recv header, 25 bytes (0x19)
0000: Framework: UltraCool 0.1.
== Info: no chunk, no close, no size. Assume close
to signal end
<= Recv header, 1 bytes (0x1)
0000: .
<= Recv data, 17 bytes (0x11)
0000: Hello from Ruby!.
== Info: Closing connection 0
```

There's a lot of fascinating stuff there. Let's talk about it.

It tells you exactly how big the header is -- 81 bytes. That number 0x51 is hexadecimal (base-16 numbers), so 5 * 16 + 1, which is also 81 bytes. All of your headers are show, with byte offsets on the

left. So the Host header starts after 19 (1*16 + 3) bytes of other headers, it looks like.

And you can see it receiving headers ***from*** your server, too. Lots of interesting stuff there.

Do you not already know about hexadecimal? Perhaps you're one of today's lucky 10,000? Check the history chapter in the back for a run-down of what hexadecimal is all about and why we use it.

If you love hexadecimal (who doesn't?), you can use --trace rather than --trace-ascii, and it will show you all the same things, plus the byte-by-byte breakdown of everything you sent. That's where you can see things like...

Have you been enjoying Rebuilding HTTP? This is the end of the free chapters. You can purchase the full version if you like! I suggest you get your employer to reimburse you or purchase it for you if you work as a software developer. After this page there are still exercise answers and historical notes!

Answers to Exercises

The exercises start simple and well-defined, and get more open-ended as you go along. That's on purpose, and I think it's a very good thing. As a result you may disagree with some of my answers, especially later on. And you **certainly** may have a different answer.

Good! It would be a boring world if we all agreed on everything.

These are *some* answers. I think they're *good* answers. But don't take them too seriously as the *only right* answers.

Chapter 1

Exercise One

I used this command: "curl -X DISCO_FEVER https://google.com/".

At least right now, Google tells me DISCO_FEVER is inappropriate, which makes me chuckle.

```
<p>The request method <code>DISCO_FEVER</code> is
inappropriate for the URL <code>/</code>.
<ins>That's all we know.</ins>
```

It's good to enjoy the little things in life.

Exercise Two

For this one, try the instructions if you can. Not everybody has two computers, un-firewalled, that can see each other. You can sometimes use a local IP address (something like: 192.168.178.75)

if you have two computers on the same local network, even if you don't have a DNS name and an external Internet address.

But for this one, it's best-effort. You may not be able to get it to work. Sometimes it's hard. Heck, sometimes you only have access to one computer.

But it's thrilling when you can see it work.

Chapter 2

Exercise One

This is just a "follow the directions" exercise. No answer needed.

Exercise Two

This is just a "follow the directions" exercise. No answer needed.

Exercise Three

This is just a "follow the directions" exercise. No answer needed.

Chapter 3

Exercise One

An easy way to get another status code besides 200 (OK) or 405 (Method Not Allowed) is to give a bad URL, something like this:

```
curl -v localhost:4322/some/random/url
```

That should give a 404. You can potentially try other things if you're creative. Can you read-protect a file so webrick isn't allowed to see it and get a 403 (Forbidden)? A custom method with a space will give a 400 (Bad Request) since it can't be parsed.

There are many other ways to get curl to generate a request that the mini-server doesn't like.

Exercise Two

There's room for variation depending what headers you set and how. In initialize you can set up default headers:

```
# Last line in RHTTP::Response#initialize
@default_h = { 'Framework': 'RebuildingHTTP' }
```

Then when you serialise the response (convert it to output,) you'll need to use them:

```
require "time" # For time#httpdate

def to_s
  h = { 'Date': Time.now.httpdate }.
    merge(@default_headers).
    merge(@headers)
  <<RESPONSE
HTTP/#{@version} #{@status} #{@message}
#{h.map {|k,v| "#{k}: #{v}" }.join("\n")}

#{@body}
RESPONSE
end
```

Now if you use an unmodified my_server.rb, you'll get the date and the default 'Framework' header. But you can also override either one with the 'headers' keyword argument:

```
# in my_server.rb, inside the loop
resp = RHTTP::Response.new("Hello from Ruby!",
  headers: { 'Framework': 'UltraCool 0.1',
    'Date': (Time.now - (60 * 60 * 24 *
365)).httpdate })
```

Try running it with and without the overrides in my_server.rb and you should see the values change in the output of curl -v.

Exercise Three

For this one, just follow the directions and compare the headers. Things you may notice include what browser is being used, what operating system the request is coming from and what language it thinks you speak. Also, notice that curl's default set of headers is much smaller.

Historical Accidents, Background, and Stories

Throughout the book, there will be times that something's weird. And I don't really want to stop for a page or more to explain why. So instead, you've been referred back here to the background.

None of the stories are important, exactly. If you know how to handle the weird case you're fine to write HTTP. But it always bothers me when I don't know why something is the way it is. Maybe you feel the same?

Welcome. Let's read some silly stories. They also happen to be about how HTTP got that way.

I may use some weird words here, or talk about weird things. Google can help you out if you need to know what hexadecimal is, or what's a teletype, or what you could use besides ASCII (ugh, EBCDIC.) The internet is stuffed with old computer history.

Why are HTTP Newlines \n\r?

First, let's talk a bit about ASCII.

ASCII is the way modern computers map from byte number values into letters. For instance, 65 is a capital "A" and 66 is a capital "B" and so on up through 90, which is "Z". 255 is the largest number a byte can hold, and often you'll see "7-bit ASCII" which only uses numbers up until 127, which means the highest bit—the 8th bit—is always zero. You'll often see these written in base-16 (a.k.a. hexadecimal) so you might see 0x41 for "A", which just means 4 * 16 + 1, or 65. So it's the same number, just written differently.

ASCII isn't just letters, though. It's also a lot of other instructions to tell a console, or a printer, what to do. For instance, you can tell it to beep, or to move to the next line, or to move in by a "tab", which is still a bitterly-debated unit of length to this very day.

We mostly use ASCII on computer screens. But the things in it are really designed for typewriters and teletype machines. Letters, sure, but also instructions to move around where you're printing.

When you want to start on a new paragraph, you need to do two things. You need to move down a line, and move the cursor back to the left side. In the old days, you had to tell the console or printer to do that. And those two things used to be separate, so you had to include both separately. The "next line" one was called "newline" and we now write it backslash-N. The "move left" one was called "carriage return" because it would move the part that printed letters (the "carriage") to where it started out ("return.") And we now write it backslash-R, for "return." So "\r\n" means "move the place we're printing to the left side and down a line."

You'll also see it called a CRLF — "carriage return, line feed." In this case, "line feed" is just another name for the newline. "Line feed" means the printer should "feed" one line worth of paper in.

There are a few odds and ends from this time you'll still see around, like "form feed" as the Ruby escape character backslash-F. A "form feed" just means "next page," but you don't normally want to do that when you're printing to console.

These days on Mac and Unix, they just use the \n to do ***both*** the next line and the carriage return. Mac and Unix figured, "you always want to do both, let's just use one character." Several other ancient operating systems, most notably Windows… did not. Windows still uses \r\n newlines. So the other common place you'll see them is if you open files generated on Windows that you've copied to your Mac or Linux machine.

HTTP was intended to be for everybody. Windows looked like it was going to rule the world forever back in 1989 when the web first happened. Windows-style newlines are harmless if you just print them directly—they're what Windows requires, and they're fine on Mac and Linux. So HTTP uses those. And you'll see them sometimes when, for instance, printing out a string with .inspect in Ruby.

Why are Windows-style newlines harmless on Mac and Unix? Because if \n moves us down a line and the cursor to the far left, and \r moves the cursor to the far left, we're still getting the same result. Moving left twice doesn't change anything. So: weird but harmless.

There are even a few old systems that use \n\r newlines. I hope you never need to know about those. For this book you can certainly ignore them.

What is Hexadecimal and Why Bother?

The short version goes: "hexadecimal is base 16 for numbers, just like binary is base 2." We can do better than that.

First off, hexadecimal is often called "hex" by its friends. Let me explain why you want to be its friend.

In day-to-day base 10 numbers ("decimal"), we count from 0 to 9. And then we add a digit: ten is written 10, or one-zero. We add a third digit when we go from 99 to 100. So far, so good.

In base 16, we count from 0 to F. So 0 to 9 are the same as in decimal, and then A is 10, B is 11, C is 12 up to F, which is 15. Lowercase letters are the same as uppercase. Sure, fine.

In base 10, to figure out the value of a number like 531, you add 5 * 100 plus 3 * 10 plus 1. The reason you use 100 is that it's 10 * 10. After 100 is 1,000, then 10,000, then 100,000 and so on.

In base 16, it goes 1 -> 16 -> 256 instead of 1 -> 10 -> 100. So you multiply by 16 every time.

Okay, sure, ***but why?*** Why bother with all this?

Computers use zeroes and ones for numbers because deep inside, computers are made of electrical wires carrying current. One voltage is zero, another voltage is one. That turned out to be an easy way to pack a lot of wires into as tiny an area as possible.

Processors can run a ***lot*** of wires through side-by-side. But the wires need some kind of grouping. Just having 500 wires that are either 0 or 1 gets hard to manage, fast.

Many years ago, after long argument and much trial and error, we wound up with groups of eight binary digits in our processors. These groups are called bytes. Eight bits means you can represent a number between 0 and 255, if you do it the obvious way. Some processors use more bits - often 32 bits or 64 bits. But notice that those are multiples of 8, and we talk about them in groups of 8.

If you want to show what's in a chunk of computer memory, you could make a long list of numbers between 0 and 255. But software developers are... ***quirky*** people. And computer monitors used to be bad, low-resolution and cheap.

If you write a number between 0 and 255 in hex then it's only ***two*** digits. That takes less space on your monitor. And sometimes you want to know what's in a single bit, not a whole byte. You can figure out each bit pretty quickly, because one hex digit is the low four bits and the other hex digit is the higher four bits. So converting from hex to binary is really easy... At least, compared to converting from decimal.

When software dumps out big chunks of data, it often does it in hexadecimal. In fact, the type of program that takes those dumps and edits them is usually called a "hex editor," even though the

editors will cheerfully show you non-hex output. But they assume you'll mostly work in hexadecimal because it's so convenient.

Appendix: Installing Ruby, Git, Bundler and SQLite3

Ruby

Any recent version of Ruby should be fine. You're welcome to install it through RVM or ruby-build. But they can be complicated, so I'm not recommending that if you don't already.

Mac OS X

To install Ruby on Mac OS X, you can first install Homebrew ("http://mxcl.github.com/homebrew/") and then "brew install ruby". For other ways, you can Google "mac os x install ruby".

Ubuntu Linux

To install Ruby on Ubuntu Linux , use apt-get:

```
> sudo apt-get install ruby-dev
```

This should install Ruby.

Others

Windows is going to be a bit hard here, though WSL (Windows System for Linux) can work. But I don't necessarily recommend Windows in any form for this book.

Google for "install Ruby on <my operating system>". If you're using an Amiga, email me!

Git (Source Control)

Mac OS X

To install git on Mac OS X if you don't have it, download and install the latest version from "http://git-scm.com/". You won't see an application icon for git, which is fine - it's a command-line application that you run from the terminal. You can also install through Homebrew.

Ubuntu Linux

To install git on Ubuntu Linux, use apt.

```
> sudo apt-get install git-core
```

Others

Google for "install git on <my operating system>".

Bundler

Bundler is a gem that Rails uses to manage all the various Ruby gems that a library or application in Ruby uses these days. The number can be huge, and there wasn't a great way to declare them before Gemfiles, which come from Bundler.

Bundler is automatically installed on the most recent Ruby versions. But you may need to install it on older Ruby. You can type "which bundle" to see if it's installed, or type "bundle -v" and see if it finds anything.

To install bundler:

```
> gem install bundler
Fetching bundler-2.3.22.gem
Successfully installed bundler-2.3.22
Parsing documentation for bundler-2.3.22
Installing ri documentation for bundler-2.3.22
Done installing documentation for bundler after 0
seconds
1 gem installed
```

Other gems will be installed via Bundler later. It uses a file called a Gemfile that just declares what gems your library or app uses, and where to find them.

SQLite

Mac OS X

Mac OS X ships with SQLite3. If the SQLite3 gem is installed correctly it should use it without complaint.

Ubuntu Linux

You'll want to use apt-get (or similar) to install SQLite. Usually that's:

```
> sudo apt-get install sqlite3 libsqlite3-dev
```

Others

Google for "install sqlite3 on <my operating system>".

Other Rubies

If you're adventurous, you know about other Ruby implementations (e.g. JRuby, Rubinius). You may need to adjust some specific code snippets if you use one.