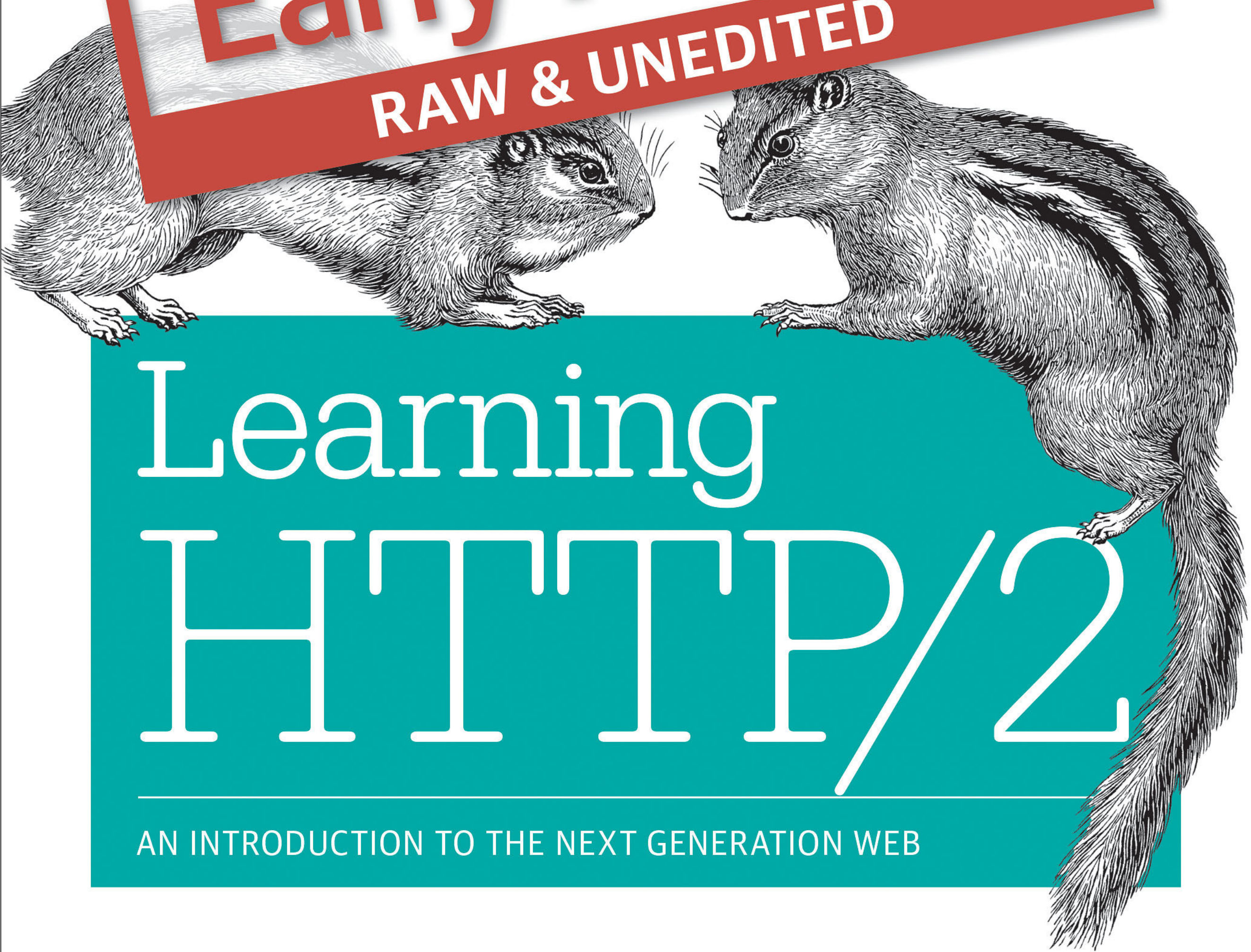
O'REILLY®

Early Release

RAW & UNEDITED



Learning HTTP/2

AN INTRODUCTION TO THE NEXT GENERATION WEB

Stephen Ludin & Javier Garza

Learning HTTP/2

Stephen Ludin and Javier Garza

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Learning HTTP/2
by Stephen Ludin and Javier Garza

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editors: Brian Anderson and Virginia Wilson
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition
2016-12-05: First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491962404* for release details.

978-1-491-96240-4

[FILL IN]

Table of Contents

source-highlighter: highlight.js :numbered!:

[[For the Impatient]]

CHAPTER 1

HTTP/2 Quick Start

When faced with something new and shiny, rarely do we want to spend the first hours meticulously going through the manual, reading the instructions, maintenance details, and safety advisories. We want to tear it out of the packaging, plug it in, turn it on, and start experiencing the wonders promised on the box. HTTP/2 (h2) should be no different.

So let's start tinkering.

Up and Running

Realistically, you have likely been experiencing HTTP/2 on a daily basis. Open a modern browser (Edge, Safari, Firefox, Chrome) and point it at a major web site (Google, Facebook, Twitter) and voila! you are using HTTP/2. Truth be told, this is likely anticlimatic and not the reason you are holding this book in your hands. Let's get things up and running so that *you* can be running the next major web site.

There are two major steps to getting an h2 server up and running:

- Get and install a web server that speaks h2
- Get and install a TLS certificate so the browser will speak h2 with the server

Neither of these are trivial, but we are going to try and make it as simple as possible. There is a more exhaustive treatment of the subject in ???, but hopefully you are running an h2 server by the end of this chapter.

Get a Certificate

Working with certificates is a subject that merits a book of its own. We are going to skip right through all of the theory and try and get a certificate in your hands for experimentation purposes ASAP. We will explore three methods: using online resources, creating a cert on your own, and obtaining a cert from Let's Encrypt. It should be noted that the first two methods will create what is called a self-signed certificate and are useful for testing purposes only.

Use an Online Generator

There are a number of resources online for generating a self-signed certificate. Since you will not have generated the private key in your own secure environment, these certs should never be used for any purpose beyond experimentation like we are doing here. A web search will quickly get you to a couple of resources. One example can be found at www.sslchecker.com[1].

Use the tool and save the generated certificate and key to two local files. Use `privkey.pem` and `cert.pem` respectively for now.

Self Signed

The openssl[2] tool is fairly universal adn easily obtainable. There are ports for almost evey major platform and it is what we will use to create our keys. If you have a unix/linux or OS X (Mac) flavor machine you very likely have it installed already. Fire up your terminal and repeat after me:

```
$ openssl genrsa -out key.pem 2048
$ openssl req -new -x509 -sha256 -key privkey.pem -out cert.pem -days 365 -subj \
        "/CN=fake.example.org"
```

With this you will have a new key called `provkey.pem` and a new cert called `cert.pem` useful for our testing.

Let's Encrypt

Let's Encrypt is a new player on the Certificate Authority scene having gone live with their public beta in the fall of 2015. Their goal is to make TLS certificates availble in an easy, automated, and inexpensive (free) manner to anyone and everyone. This is core to the *TLS Everywhere* movement which can be summed up the belief that all of our web communications should always be encrypted and authenticated. For our purposes here, the "easy" bit is what is attractive so we can get up and running as soon as possible.

1 *https://www.sslchecker.com/csr/self_signed*

2 *https://www.openssl.org*

Though there are now many clients and libraries to choose from[3] The Electronic Frontier Foundation (EFF) maintains the Let's Encrypt recommended client called certbot[4]. Certbot is intended to make certificate acqusition and maintenance a complete hands off process doing everythign from obtaining the certificate and then installing the certificate on your webserver for you.

In order to obtain a certificate ftom Let's Encrypt you will need to be able to validate your domain with them. This implies you have control of the domain and can prove it by modifying DNS or the Webserver to prove it to Let's Encrypt. If you do not have a domain or do not want to be bothered, simply use on of the *self signed* method above.

Follow instructions for downloading certbot for your favorite operating system. For the purposes of this chapter you do not need to be concerned with the webserver choice. For linux flavors, the simplest method for most cases is to do the following on the machine running your webserver:

```
$ wget https://dl.eff.org/certbot-auto
$ chmod a+x certbot-auto
```

Once downloaded, run `certbot-auto` like:

```
$ ./certbot-auto certonly --webroot -w <your web root> -d <your domain>
```

substituting your webserver filesystem root and your domain in the relevant places. This will automatically install any needed packages, prompt you with a number of questions, and finally if all goes well obtain the certificate from Let's Encrypt. Your newly minted cert and private keys will be placed in `/etc/letsencrypt/live/<your domain>`:

File	Description
/etc/letsencrypt/live/<your domain>/privkey.pem	Your certificate's private key
/etc/letsencrypt/live/<your domain>/cert.pem	Your new certificate
/etc/letsencrypt/live/<your domain>/chain.pem	The Let's Encrypt CA Chain
/etc/letsencrypt/live/<your domain>/fullchain.pem	Your new cert and the chain all in one

You will be using these files in the next step.

3 *https://community.letsencrypt.org/t/list-of-client-implementations/2103*

4 *https://certbot.eff.org/*

Get and Run Your First HTTP/2 Server

There are numerous choices already out there for obtaining and running a webserver that speaks HTTP/2. ??? goes into a fairly comprehensive list in some detail. Our goal here is quick and simple and for that we will look towards the `nghttp2` package. `nghttp2` [5], developed by Tatsuhiro Tsujikawa, provides a number of useful tools for working with and debugging HTTP/2. For now, we are interested in the `nghttpd` tool.

There is more information on installing nghttp2 in ??? but we'll hit the high points here. Install nghttp2 via your favorite package manager or (for the brave) from the source. For example, on Ubuntu 16:

```
$ sudo apt-get install nghttp2
```

Once installed, with certificates in hand, run nghttpd as

```
$ ./nghttpd -v -d <webroot> <port> <key> <cert>
```

Where `<webroot>` is the path to your website, `<port>` is the port you want the server to listen to, and `<key>` and `<cert>` are paths to the key and certificate you generated. For eaxmple:

```
$ ./nghttpd -v -d /usr/local/www 8443 /etc/letsencrypt/live/yoursite.com/privkey.pem \
                    /etc/letsencrypt/live/yoursite.com/cert.pem
```

Pick a Browser

Finally, the reward for the hard work. Pick a modern browser and point it at your new server. See ??? for a very comprehensive list of browsers. If you created a self signed certificate you should see a security warning. Confirm that it is complaining about the cert you created and accept the warnings. You should see your website now.

And it is being served over h2!

5 *https://nghttp2.org*

CHAPTER 2
Evolution of HTTP

In the 1930s Vannevar Bush, an electrical engineer from the United States then at MIT's School of Engineering, had a concern with the volume of information we were producing relative to society's ability to consume that information. In his essay published in the Atlantic Monthly in 1945 entitled, "As We May Think," he said:

> Professionally our methods of transmitting and reviewing the results of research are generations old and by now are totally inadequate for their purpose. If the aggregate time spent in writing scholarly works and in reading them could be evaluated, the ratio between these amounts of time might well be startling.
>
> —Vannevar Bush, *Atlantic Monthly*

He envisioned a system where our aggregate knowledge was stored on microfilm and could be "consulted with exceeding speed and flexibility." He further stated that this information should have contextual associations with related topics, much in the way the human mind links data together. His *memex* system was never built, but the ideas influenced those that followed.

The term *Hypertext* that we take or granted today was coined around 1963 and first published in 1965 by Ted Nelson, a software designer and visionary. He proposed the idea of hypertext:

> to mean a body of written or pictorial material interconnected in such a complex way that it could not conveniently be presented or represented on paper. It may contain summaries, or maps of its contents and their interrelations; it may contain annotations, additions and footnotes from scholars who have examined it. [1]
>
> —Ted Nelson

1 T. H. Nelson, "Complex information processing: a file structure for the complex, the changing and the indeterminate", ACM '65 Proceedings of the 1965 20th national conference

Nelson wanted to create a “docuverse” where information was interlinked and never deleted and easily available to all. He built on Bush’s ideas and in the 1970s created a prototype implementations of his project Xanadu. It was unfortunately never completed, but provided the shoulders to stand on for those to come.

HTTP enters the picture in 1989. While at CERN, Tim Berners-Lee proposed [2] a new system for helping keep track of the information created by the accelerators (referencing the yet to be built Large Hadron Collider) and experiments at the institution. He embraces two concepts from Nelson: Hypertext, or “Human-readable information linked together in an unconstrained way,” and Hypermedia a term to “indicate that one is not bound to text.” In the proposal he discussed the creation of a server and browsers on many machines to provide a “universal system.”

HTTP/0.9 and 1.0

HTTP/0.9 was a wonderfully simple, if limited, protocol. It had a single method (GET), there were no headers, and it was designed to only fetch HTML (meaning no images - just text).

Over the next few years use of HTTP grew across the world. By 1995 there were over 18,000 servers handling HTTP traffic on port 80 across the world. The protocol had evolved well past its 0.9 roots and in 1996 RFC 1945[3] codified HTTP/1.0.

Version 1.0 brought a massive amount of change to the little protocol that started it all. Whereas the 0.9 *spec* was about a page long, the 1.0 RFC measured in at 60 pages. You could say it had grown from a toy into a tool. It brought in ideas that are very familiar to us today:

- Headers
- Response Codes
- Redirects
- Errors
- Conditional request
- Content Encoding (compression)
- More request methods

and more. HTTP/1.0, though a large leap from 0.9 still had a number of known flaws to be addressed. Most notably were the inability to keep a connection open between

2 *https://www.w3.org/History/1989/proposal.html*

3 “https://tools.ietf.org/html/rfc1945”

requests, the lack of a mandatory Host header, and bare bones options for caching. These two items had consequences on how the web could scale and needed to be addressed.

HTTP/1.1

Right on the heels of 1.0 came 1.1, the protocol that has lived on for over 20 years. It fixed a number of the aforementioned 1.0 problems. By making the Host header mandatory, it was now possible to perform *virtual hosting* or serving multiple web properties on a singe IP address. When the new connection directives are used, a web server was not required to close a connection after a response. This was a boon for performance and efficiency since the browser no longer needed to reestablish the TCP connection on every request.

Additional changes included:

- An extension of cachability headers
- An OPTIONS method
- The Upgrade header
- Range requests
- Compression with Transfer-encoding
- Pipelining
- And much much more

Pipelining is a feature that allows a client to send all of its request at once. This may sound a bit like a preview of *Multiplexing* which will come in HTTP/2. There were a couple of problems with pipelining that prevented its popularity. Servers still had to respond to the requests in order. This meant if one request takes a long time, this *head of line blocking* will get in the way of the other requests. Additionally pipelining implementations in servers and proxies on the internet tended to range from nonexistant (bad) to broken (worse).

HTTP/1.1 was the result of HTTP/1.0's success and the experience gained running the older protocol for a few years.

RFCs for HTTP/1.1

The Internet Engineering Task Force (IETF) publishes protocol specifications in committee created drafts called *Request For Comments* (RFC). These committees are open to anyone with the time and inclination to participate. HTTP/1.1 was first defined in RFC 2068, then later replaced by RFC 2616, and finally revised in RFCs 7230 through 7235.

Beyond 1.1

Since 1999 RFC 2616, which then specified HTTP/1.1, has defined the standard that the modern web is built on. Written in stone, it did not evolve or change. The web, however, and the way we used it continued to change in way likely unimagined by its originators. The interactivity and utility of your average commerce site goes well beyond the vision of an interwoven docuverse and fundamentally changes the way we participate in our world. That evolution came despite the limitation of the protocol that we see today.

The most tangible change we can point to is in the makeup of the web page. The HTTP Archives only goes back to 2010, but in even that relatively short time the change has been dramatic. Every added object adds complexity and strains a protocol designed to request one object at a time.

SPDY

In 2009, Mike Belshe and Roberto Peon of Google proposed an alternative to HTTP which they called SPDY (Pronounced SPeeDY)[4]. SPDY was not the first proposal to replace HTTP, but it was the most important as it moved the perceived mountain. Before SPDY, it was thought that there was not enough will in the industry to make breaking changes to HTTP/1.1. The effort to coordinate the changes between browsers, servers, proxies, and various middle boxes was seen to be too great. But SPDY quickly proved that there was a desire for something more efficient and a willingness to change.

SPDY laid the groundwork for HTTP/2 and was responsible for proving out some of its key features such as multiplexing, framing, and header compression amongst others.It was integrated in relative speed into Chrome and Firefox and eventually would be adopted by almost every major browser. Similarly, the necessary support in servers and proxies came along at about the same pace. The desire and the will were proven to be present.

4 *http://dev.chromium.org/spdy/spdy-protocol/spdy-protocol-draft1*

HTTP/2

In early 2012, the HTTP Working Group, the IETF group responsible for the HTTP specifications, was rechartered to work on the next version of HTTP. A key portion of their charter laid out their expectations for this new protocol:

> It is expected that HTTP/2.0 will:
>
> - Substantially and measurably improve end-user perceived latency in most cases, over HTTP/1.1 using TCP.
> - Address the "head of line blocking" problem in HTTP.
> - Not require multiple connections to a server to enable parallelism, thus improving its use of TCP, especially regarding congestion control.
> - Retain the semantics of HTTP/1.1, leveraging existing documentation (see above), including (but not limited to) HTTP methods, status codes, URIs, and where appropriate, header fields.
> - Clearly define how HTTP/2.0 interacts with HTTP/1.x, especially in intermediaries (both 2->1 and 1->2).
> - Clearly identify any new extensibility points and policy for their appropriate use.[5]

A call for proposals was sent out and it was decided to use SDPY as a starting point for HTTP/2.0. Finally, on May 14, 2015 RFC 7540 was published and HTTP/2 was official.

The remainder of this book lays out the rest of the story.

5 *https://datatracker.ietf.org/wg/httpbis/charter/*