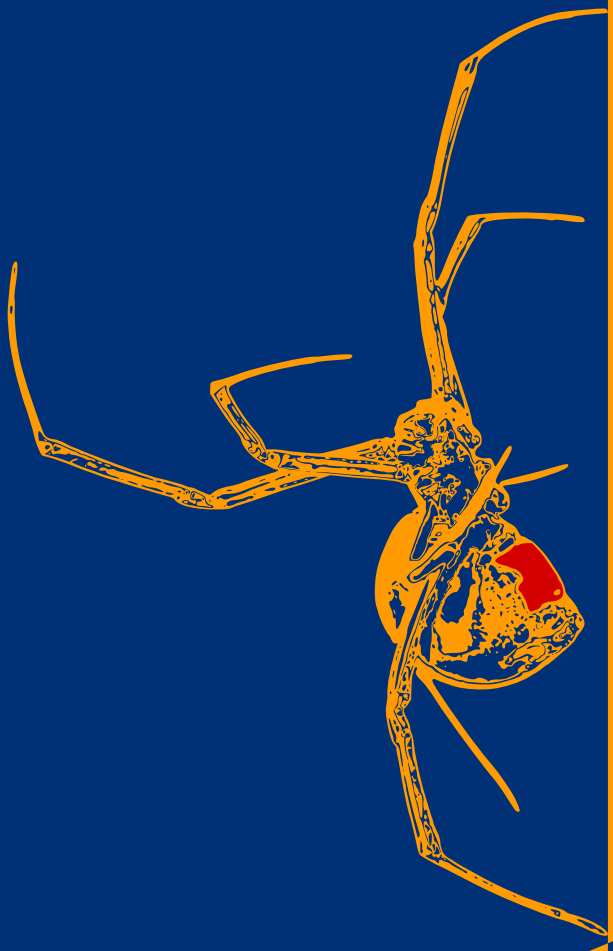
Project-Based Learning

Recreate critical tech
Master fundamentals

Build Your Own Web Server From Scratch in Node.JS

Network protocol
HTTP in detail
Concurrency
WebSocket



TypeScript

James Smith
build-your-own.org

Build Your Own Web Server From Scratch in Node.JS

Learn network programming, the HTTP protocol, and its applications by coding your own Web Server.

James Smith

2024-02-02 build-your-own.org

Contents

CHAPTER 01

Introduction

1.1 Why Code a Web Server?

Most people use HTTP daily, but few understand its inner workings. This "Build Your Own X" book dives deep, teaching basics from scratch for a clearer understanding of the tools and tech we rely on.

Network Programming

The first step is to make programs talk over a network. This is also called *socket programming*. But socket programming is more than just gluing APIs together! It's easy to end up with half-working solutions if you skip the basics.

Protocols & Communication

In order to communicate over a network, the data sent over the network must conform to a specific format called a "protocol". Learn how to create or implement any network protocols by using HTTP as the target.

HTTP in Detail

You probably already know something about HTTP, such as URLs, different methods like GET and POST, response codes, various headers, and etc. But have you ever thought that you can create all the details from scratch, by your own code? It's not very complicated and it's rewarding.

1.2 Build Your Own X From Scratch

Why take on a build-your-own-X challenge? A few scenarios to consider:

- Students: Solidify learning, build portfolio, stand out in future careers.
- Developers: Master fundamentals beyond frameworks and tools.
- Hobbyists: Explore interests with flexible, extensible projects.

1.3 The Book

Project-Centric

Designed to guide you through your own web server implementation, the book follows a *step-by-step* approach, and each chapter builds on the previous one.

An outline of each step:

1. TCP echo server.
    - Socket primitives in Node.JS.
    - Event-based programming.
    - `async/await` and promises.
2. A simple messaging protocol.
    - Implementing a network protocol in TCP.
    - Working with byte streams and managing buffers.
3. Basic HTTP server.
    - HTTP semantics and syntax.
    - Generating dynamic content.
        - Async generators.
4. Core applications.
    - Serving static files.
        - File IO in Node.JS.
        - Programming tip: avoiding resource leaks.
        - Range requests.
    - Caching control.
    - Compression.
        - The stream and pipe abstraction.
5. WebSocket.
    - Message-based designs.
    - Concurrent programming.
        - Blocking queues.

Producing code is the easiest part and you'll also need to *debug* the code to make it work. So I have included some tips on this.

Discussions at the End of Chapter

Getting your own web server running is rewarding, but it should not be your only goal. There are many blog posts that show you a toy HTTP server. But how do you get beyond toys?

At the end of each chapter, there are discussions about:

- What's missing from the code? The gap between toys and the real thing, such as optimizations and applications.
- Important concepts beyond coding, such as event loops and backpressure. These are what you are likely to overlook.
- Design choices. Why stuff has to work that way? You can learn from both the good ones and the bad ones.
- Alternative routes. Where you can deviate from this book.

Node.js and TypeScript

The project uses Node.js without any dependencies, but many concepts are *language-agnostic*, so it's valuable for learners of any language.

Code samples use TypeScript with type annotations for readability, but the differences from JS are minor.

The code is mostly *data structures* + *functions*, avoiding fancy abstractions to maximize clarity.

Book Series

This is part of the "*Build Your Own X*" book series, which includes books on building your own Redis[1], database[2], and compiler[3].

https://build-your-own.org

[1] https://build-your-own.org/redis/
[2] https://build-your-own.org/database/
[3] https://build-your-own.org/compiler/

CHAPTER 02

HTTP Overview

2.1 Overview

As you may already know, the HTTP protocol sits above the TCP protocol. How TCP itself works in detail is not our concern; what we need to know about TCP is that it's a bidirectional channel for transmitting raw bytes — a carrier for other application protocols such as HTTP or SSH.

Although each direction of a TCP connection can operate independently, many protocols follow the request-response model. The client sends a request, then the server sends a response, then the client might use the same connection for further requests and responses.

```
 client        server
 ------        ------

| req1 |  ==>
         <==  | res1 |
| req2 |  ==>
         <==  | res2 |
         ...
```

An HTTP request or response consists of a *header* followed by an optional *payload*. The header contains the URL of the request, or the response code, followed by a list of *header fields*.

2.2 HTTP by Example

To get the hang of network protocols, let's start by making an HTTP request from the command line.

Run the netcat command:

```
nc example.com 80
```

The `nc` (netcat) command creates a TCP connection to the destination host and port, and then attaches the connection to stdin and stdout. We can now start typing in the terminal and the data will be transmitted:

```
GET / HTTP/1.0
Host: example.com
(empty line)
```

(Note the extra empty line at the end!)

We will get the following response:

```
HTTP/1.0 200 OK
Age: 525410
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Thu, 20 Oct 2020 11:11:11 GMT
Etag: "1234567890+gzip+ident"
Last-Modified: Thu, 20 Oct 2019 11:11:11 GMT
Vary: Accept-Encoding
Content-Length: 1256
Connection: close

<!doctype html>
<!-- omitted -->
```

Making HTTP requests from the command line is very easy. Let's take a look at the data and try to figure out what it means.

The first line of the request — `GET / HTTP/1.0` — contains the HTTP method `GET`, the URI `/`, and the HTTP version `1.0`. This is easy to figure out.

And the first line of the response — `HTTP/1.0 200 OK` — contains the HTTP version and the response code `200`.

Following the first line is a list of header fields in the format of `Key: value`. The request has a single header field — `Host` — which contains the domain name.

The response contains many fields, and their functions are not as obvious as the `Host` field. Many HTTP header fields are optional, and some are even useless. We will learn more about this in later chapters.

The response header is followed by the payload, which in our case is an HTML document. Payload and header are separated by an empty line. The GET request has no payload so it ends with an empty line.

This is just a simple example that you can play with from the command line. We will examine the HTTP protocol in more detail later.

2.3 The Evolution of HTTP

HTTP/1.0: The Prototype

The example above uses HTTP/1.0, which is an ancient version of HTTP. HTTP/1.0 doesn't support multiple requests over a single connection at all, and you need a new connection for every request. This is problematic because typical web pages depend on many extra resources, such as images, scripts, or stylesheets; the latency of the TCP handshake makes HTTP/1.0 very suboptimal.

HTTP/1.1 fixed this and became a practical protocol. You can try using the `nc` command to send multiple requests on the same connection by simply changing `HTTP/1.0` to `HTTP/1.1`.

HTTP/1.1: Production-Ready & Easy-to-Understand

This book will focus on HTTP/1.1, as it is still very popular and easy to understand. Even software systems that have nothing to do with the Web have adopted HTTP as the basis of their network protocol. When a backend developer talks about an "API", they likely mean an HTTP-based one, even for internal software services.

Why is HTTP so popular? One possible reason is that it can be used as a generic request-response protocol; developers can reply on HTTP instead of inventing their own protocols. This makes HTTP a good target for learning how to build network protocols.

HTTP/2: New Capacities

There have been further developments since HTTP/1.1. HTTP/2, related to SPDY, is the next iteration of HTTP. In addition to incremental refinements such as compressed headers, it has 2 new capacities.

- *Server push*, which is sending resources to the client before the client requests them.
- *Multiplexing* of multiple requests over a single TCP connection, which is an attempt to address *head-of-line blocking*.

With these new features, HTTP/2 is no longer a simple request-response protocol. That's why we start with HTTP/1.1 because it's simple enough and easy to understand.

HTTP/3: More Ambition

HTTP/3 is much larger than HTTP/2. It replaces TCP and uses UDP instead. So it needs to replicate most of the functionality of TCP, this TCP alternative is called QUIC. The motivations behind QUIC are userspace congestion control, multiplexing, and head-of-line blocking.

You can learn a lot by reading about these new technologies, but you may be overwhelmed by these concepts and jargon. So let's start with something small and simple: coding an HTTP/1.1 server.

2.4 Command Line Tools

You are introduced to the command line because it allows for quick testing and debugging. which is handy when coding your own HTTP server. Although you may want to store the request data in a file instead of typing it each time.

```
nc example.com 80 <request.txt
```

In practice, you may encounter some quirks of the **nc** command, such as not sending EOF, or multiple versions of **nc** with incompatible flags. You can use the modern replacement **socat** instead.

```
socat tcp:example.com:80 -
```

The **telent** command is also popular in tutorials.

```
telnet example.com 80
```

You can also use an existing HTTP client instead of manually constructing the request data. Try the **curl** command:

```
curl -vvv http://example.com/
```

Most sites support HTTPS alongside plaintext HTTP. HTTPS adds an extra protocol layer called "TLS" between HTTP and TCP. TLS is not plaintext, so you cannot use netcat to test an HTTPS server. But TLS still provides a byte stream like TCP, so you just need to replace netcat with a TLS client.

```
openssl s_client -verify_quiet -quiet -connect example.com:443
```

In the next chapter, we will do some actual coding.

CHAPTER 03

Code A TCP Server

Our first step is to get familiar with the socket API, so we will code a simple TCP server in this chapter.

3.1 TCP Quick Review

Layers of Protocols

Network protocols are divided into different layers, where the higher layer depends on the lower layer, and each layer provides different capacities.

```
 top
  /\    | App |     message or whatever
  ||    | TCP |     byte stream
  ||    | IP  |     packets
  ||    | ... |
bottom
```

The layer below TCP is the IP layer. Each IP packet is a message with 3 components:

- The sender's address.
- The receiver's address.
- The message data.

Communication with a packet-based scheme is not easy. There are lots of problems for applications to solve:

- What if the message data exceeds the capacity of a single packet?
- What if the packet is lost?
- Out-of-order packets?

To make things simple, the next layer is added on top of IP packets. TCP provides:

- Byte streams instead of packets.
- Reliable and ordered delivery.

A byte stream is simply an ordered sequence of bytes. A *protocol*, rather than the application, is used to make sense of these bytes. Protocols are like file formats, except that the total length is unknown and the data is read in one pass.

UDP is on the same layer as TCP, but is still packet-based like the lower layer. UDP just adds port numbers over IP packets.

TCP Byte Stream vs. UDP Packet

The key difference: boundaries.

- UDP: Each *read* from a socket corresponds to a single *write* from the peer.
- TCP: No such correspondence! Data is a continuous flow of bytes.

TCP simply has no mechanism for preserving boundaries.

1. TCP send buffer: This is where data is stored before transmission. Multiple writes are indistinguishable from a single write.
2. Data is encapsulated as one or more IP packets, IP boundaries have no relationship to the original write boundaries.
3. TCP receive buffer: Data is available to applications as it arrives.

The No. 1 beginner trap in socket programming is "concatenating & splitting TCP packets" because there is no such thing as "TCP packets". Protocols are required to interpret TCP data by imposing boundaries within the byte stream.

Byte Stream vs. Packet: DNS as an Example

To help you understand the implications of the byte stream, let's use the DNS protocol (domain name to IP address lookup) as an example.

DNS runs on UDP, the client sends a single request message and the server responds with a single response message. A DNS message is encapsulated in a UDP packet.

```
| IP header |         IP payload          |
            \.............................../
             | UDP header |  UDP payload  |
                          \.............../
                           | DNS message |
```

Due to the drawbacks of packet-based protocols, e.g., the inability to use large messages, DNS is also designed to run on TCP. But TCP knows nothing about "message", so when sending DNS messages over TCP, a **2-byte length field** is prepended to each DNS message so that the server or client can tell which part of the byte stream is which message. This 2-byte length field is the simplest example of an *application protocol* on top of TCP. This protocol allows for multiple *application messages* (DNS) in a single TCP byte stream.

```
| len1 | msg1 | len2 | msg2 | ...
```