Modern Web

Neil Kingdom

February 21, 2024

Contents

1 Preface

The purpose of this document is to give the reader a high-level overview of the plethora of advanced topics that are involved with the modern web ecosystem. Of course, I am only one person writing this, so I do not have the time or will-power to cover every concept or update this document to include every new web technology or framework that is released. For new web programmers who barely have any concept of how to write even basic HTML, I will unfortunately not be covering any language syntax, whether that be for HTML, CSS, JavaScript, Typescript, etc. The aforementioned languages are all very easy to learn and there are many great resources online if you are still just starting out. Otherwise, this document should be very accessible even to beginners.

2 Introduction

As you're likely aware, the web is one of the quickest technologies to evolve and innovate. For programmers who did not get the opportunity to follow these changes since the web's inception, it can be quite daunting to try getting into web when it seems that their already so far behind. As most probably know, the web is quite a recent phenomena, beginning in only the early 90's. We refer to this era as web 1.0. This was a time where websites were written in pure HTML and CSS. There was no concept of JavaScript or any of these other fancy technologies. It was plain and simple... Perhaps a little too plain and simple though. People could read blogs and articles and the news during this time, but it was more of a cool toy rather than something taken seriously. It was with the rise of things like MySpace and eventually Facebook where we entered web 2.0: a more interactive era of the web where users could begin to input data and receive feedback. Some argue that we are entering into the era of web 3.0 with AI, as well as WebAssembly, which will allow people to run extremely CPU intensive applications natively in a web interface/browser.

3 Revisiting Networking Protocols

In order to get to where we are today, it's best, in my opinion, to go back to basics. If you haven't already read my networking notes or have some knowledge of networking, I might recommend that as a precursor to this document. Luckily, we don't need to understand too many protocols when it comes to web development. The most important are by far DNS, TCP, HTTP, and HTTPS (which is a composition of HTTP and TLS - formerly known as SSL). In case you're not familiar with the ackronyms, TCP stands for Transmission Control Protocol. It rests at the transfer layer of both the OSI and TCP/IP models alongisde UDP, which stands for User Datagram Protocol. TCP has been the defacto standard for transferring packets across the web, although this is changing with HTTP/3. HTTP stands for HyperText Transfer Protocol and rests at the Application Layer in both the OSI and TCP/IP models. HTTP is the standard for fetching or sending data related to the actual content that is rendered on the user-side. HTTP/3 is the newest standard for HTTP and is beginning to use UDP rather than TCP. UDP is a much less reliable protocol for sending data since network packets are not acknowledged when received. They may come out of order or be dropped entirely. On the contrary, TCP packets are sent in the correct order and must be acknowledged by the receiver, otherwise the sender must re-send the lost packet. HTTPS is just HTTP "Secure". It uses the Transfer Layer Secure (TLS), formerly known as Secure Socket Layer (SSL), protocol on top of HTTP to encrypt the contents of HTTP requests and responses. Virtually all of the web uses HTTPS these days. The last really relevant protocol used in web is Domain Name System (DNS). We'll cover DNS in more detail, but know that it's essentially how your browser is able to take a URL and convert it into an IP address that it can find. Speaking of IP addresses, it is somewhat important to remember that your router uses Network Address Translation (NAT) to convert your private IP address into a public facing IP address. On Linux, you can use the ip command (probably already installed on your distro) to list out your PC's private IP address. You can either run ip addr show, or abbreviated: ip a show. This will list the loopback, ethernet, and wireless interfaces for your network card listed as something like lo, eth0, and wlan0, respectively. If not installed, get the nmap command. Copy your PC's IP address alongside the port and run the following: nmap -T4 xxx.xxx.xxx.xxx/xx. This will list all devices connected to your LAN. These should all share some

portion of your PC's IP address depending upon the subnet mask. Typically, the subnet mask defaults to 255.255.255.0, meaning that all IP addresses on your LAN will share the first three bytes xxx.xxx.xxx.nnn where nnn is unique to your device. When any device on your network requests a resource such as a webpage, the request is forwarded to your router. Your router/default gateway takes your PC's private IP address and converts it to a public-facing IP address using a NAT Translation Table before sending it to your ISP to be forwarded to the internet. There are websites such as showmyip.com that will tell you what your public-facing IP address is.

4 The HTTP Request and Response

If we're to understand web, it's quite crucial to understand HTTP requests and responses. An HTTP request/response is a protocol for HTTP servers to know how and what to fetch or send to the user. An HTTP request adheres to the following format:

Method Request-URI HTTP-Version CRLF
Headers CRLF
Message-body

Let's break this down:

- **Method:** HTTP defines certain request methods which are REST compliant. We'll talk more about REST later. These methods include GET, POST, PUT, DELETE (these are the most common ones), HEAD, CONNECT, OPTIONS, TRACE, and PATCH. GET and POST are often confused with one another. Essentially GET is used to get a new resource, whereas POST is used to create or update a resource. GET is known as an idempotent request, meaning non-destructive, whereas POST is non-idempotent, meaning that it will overwrite previous data. For example, if creating a button that increments a shopping cart count, you'd want request to be a POST request to update the count, otherwise it would remain the same if using GET. HEAD is like GET but omits the response body. PUT will replace all instances of a target resource with the request payload.
- **Request-URI:** The Request-URI is a Uniform Resource Identifer (URI) that is used to identify a name or resource on the internet. This differs from a Uniform Resource Locator (URL), which is a subset of URIs and specifies where the resource is available and the mechanism for retrieving it. A URI can be made up of the following components:
  - **Scheme/Protocol:** This is the method for fetching the resource e.g. http
  - **Authority** Consists of the host and the port number. The host is further divided into the subdomain (usually www), the domain name, and the top-level domain (e.g .org, .gov, .com, etc.). An example of an authority would be www.example.org:8080.
  - **Path** The host name resolves to the root directory on the web server. The path is a relative path from the root directory to the resource that is actually being acquired.
  - **Query** Queries are an optional list of parameters that proceed the question mark in the URI and are separated by an ampersand.
  - **Fragment** The fragment is an optional sub-resource that proceeds the hash mark.
    Example URI: http://www.example.org:8080/path/to/resource?param=value&a=5#paragraph3
- **HTTP-Version:** This will look like HTTP/n.n where n is the version number.
- **CRLF:** CRLF stands for carriage-return followed by line-feed. Linux typically only uses line-feeds for separating text (e.g. \n), but NTFS on Windows uses CRLF (\r\n) which is also what the HTTP request uses.

- **Headers:** There are a series of available headers defined by the IANA which allow the client and server to pass additional information alongside the Message-body. A common one that we care about is the accept header which tells the server about what kinds of data can be sent back.
- **Message-body:** The message body is not applicable for certain HTTP methods such as GET, but it will apply to requests such as PUT or POST. It will usually either be XML, JSON, or HTML/text data.

To provide a visual of an HTTP response we can use the curl command to fetch a website. You must run curl with the verbose option (-v) to see the HTTP request and response e.g. curl -v www.neilkingdom.xyz. In my case, this outputs the following for the HTTP request:

* Trying 149.248.53.42:80...
Connected to www.neilkingdom.xyz (149.248.53.42) port 80
> GET / HTTP/1.1
> Host: www.neilkingdom.xyz
> User-Agent: curl/8.4.0
> Accept: */*

We can see that curl made a GET request using HTTP/1.1. Since it's a get request there is no body, but we do see some headers in the form of User-Agent and Accept. A value of */* for accept indicates that the server can return anything it wants.

HTTP responses look slightly different from requests, but follow a similar pattern:

HTTP-Version Status-Code Reason-Phrase CRLF
headers CRLF
Message-body

The only thing new here are the Status-Code and Reason-Phrase. You've surely seen an HTTP status code before whether it was a 404 - not found, 200 - ok, 500 - internal server error, 301 - moved permanently, etc. HTTP status codes are split into categories based on their ranges. Here are what each range represents:

1. Informational responses (100-199) 2. Successful responses (200-299) 3. Redirection messages (300-399) 4. Client error responses (400-499) 5. Server error responses (500-599)

The Reason-Phrase is simply the string of text that describes the Status-Code. HTTP responses almost always contain a Message-body. Here is the remainder of the output of the curl command that I ran previously:

< HTTP/1.1 301 Moved Permanently
< Server: nginx/1.22.1
< Date: Sat, 09 Dec 2023 17:22:09 GMT
< Content-Type: text/html
< Content-Length: 169
< Connection: keep-alive
< Location: https://www.neilkingdom.xyz/
<
<html>
<head><title>301 Moved Permanently</title></head>
<body>
<h1>301 Moved Permanently</h1>
<hr> nginx/1.22.1
</body>
</html>

The reason I got a 301 here is actually because I didn't specify that I wanted to locate the resource using the https protocol and curl assumed I wanted http but my web server is setup to redirect to the https version. The output is easier to read this way though, since the body is concise. We see at the top our status code (301), followed by the reason phrase (Moved Permanently). Everything up until the HTML code are more HTTP headers.

5 Writing a Basic HTTP Web Server

Of course, it is best to learn by doing, and so I'm going to provide the code for writing a basic web server in C and then we'll go through it. This will be a repeat of the code and explanations provided in my POSIX Programming notes.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <limits.h>
#include <stdarg.h>
#include <errno.h>
#include <stdbool.h>
#include <arpa/inet.h>  /* Internet socket APIs */
#include <sys/socket.h> /* Berkley socket APIs */

#define SERVER_PORT 8080 /* Standard HTTP port */
#define CRLF "\r\n"

typedef struct sockaddr sockaddr_t;
typedef struct sockaddr_in sockaddr_in_t;

int main (int argc, char **argv) {
   int listen_fd, conn_fd, n;
   sockaddr_in_t serv_addr;
   uint8_t buf[LINE_MAX];
   uint8_t recv_line[LINE_MAX];

   listen_fd = socket(AF_INET, SOCK_STREAM, 0);
   if (listen_fd < 0) {
      fprintf(stderr, "Socket error\n");
      exit(EXIT_FAILURE);
   }

   serv_addr = (sockaddr_in_t){
      .sin_family       = AF_INET,
      .sin_port         = htons(SERVER_PORT),
      .sin_addr.s_addr  = htonl(INADDR_ANY)
   };
```

```
    if (bind(listen_fd, (sockaddr_t *)&serv_addr, sizeof(serv_addr)) < 0) {
        fprintf(stderr, "Bind error\n");
        exit(EXIT_FAILURE);
    }

    if (listen(listen_fd, 10) < 0) {
        fprintf(stderr, "Listen error\n");
        exit(EXIT_FAILURE);
    }

    while (true) {
        sockaddr_in_t addr;
        socklen_t addr_len;

        printf("Waiting for a connection on port %d\n", SERVER_PORT);
        conn_fd = accept(listen_fd, (sockaddr_t *)NULL, NULL);

        memset(recv_line, 0, sizeof(recv_line));

        while ((n = read(conn_fd, recv_line, LINE_MAX - 1)) > 0) {
            printf("%s\n", recv_line);

            if (strstr((const char*)recv_line, CRLF CRLF) != NULL) {
                break;
            }
        }

        if (n < 0) {
            fprintf(stderr, "Read error");
            exit(EXIT_FAILURE);
        }

        snprintf((char*)buf, sizeof(buf), "HTTP/1.1 200 OK" CRLF CRLF "Hello from
        ↪  server");

        write(conn_fd, (char*)buf, strlen((char*)buf));
        close(conn_fd);
    }

    return EXIT_SUCCESS;
}
```

In order to execute this code, compile using GCC e.g. gcc server.c -o server, then run the server. From a browser, go to localhost:8080. In case you're not aware, localhost resolves to your loopback address, which should be 127.0.0.1. As you should also know, port 80 is the standard port for HTTP and port 443 is the standard port for HTTPS. Port numbers under 1000 are reserved and can easily conflict with other applications, thus requiring superuser privileges to execute applications on. A common port that people use to avoid this conflict is 8080, but you may choose any port you'd like. You sould receive in the browser some text saying Hello from server. In the server application, you can view the HTTP requests similar to the functionality of curl. There will likely be multiple requests because your browser will try to fetch multiple things. For example, the first request for me specifies root (/) as the URI, followed by /favicon.ico as the

URI for the second request. ico files are used to display the icon on the tab for the current webpage and the default name for them is favicon.

Let's run through the code now. We create 2 file descriptors: listen_fd and conn_fd. listen_fd is set to be a TCP socket, indicated by SOCK_STREAM (UDP would be SOCK_DGRAM). The AF_INET flag means "address family internet", which marks this file descriptor as being an internet socket, as opposed to a Berkley socket if we had supplied AF_UNIX, which are used for Inter-Process Communications (not relevant to what we're talking about). We setup a socket address struct called serv_addr that contains, again, the address family, as well as the port and address range that we want to listen for. INADDR_ANY specifies that we want to listen for any connection. The htons and htonl stand for host-to-network short/long. The internet works in network-byte order which is fancy talk for big-endian, so putting our port number and ip address range in here ensures that its getting sent in the proper order (most PCs are little-endian). Next, we bind the server address to our listen file descriptor and listen for incomming traffic. Calling bind() immediately followed by listen() is almost always the use-case. Just because we're listening for connections now, doesn't mean that we're accepting them. Since we might have to handle many connections, we process each one in a loop. A more robust way to handle this sort of thing is to use a thread pool to handle each connection plus a fixed-size queue that blocks incoming connection requests when the queue is full. The accept() function blocks until a connection is established. Reading or writing to this descriptor allows us to communicate with the client. Since we know that our request will end with to CRLFs we use strstr() to check if the substring \r \n \r \n exists in the content received. Finally, we write our response "Hello from server" as plain text. We could optionally have made this XML, JSON, HTML, image data, etc. In order to write a true HTTP server, we would simply need to incorporate logic to validate the request, parse the URI, and respond with the appropriate resource.

6 Web Browsers Explained

7 Hostname Resolution With DNS

We briefly touched upon DNS, but I promised we'd revisit it in further detail. A DNS server is a server that provides DNS services (duh). The DNS server that your PC uses will initially be determined by your ISP. You are allowed to use other DNS servers. Well known ones include Google's DNS server, which is a popular address used to test ping (8.8.8.8), OpenDNS, Cloudflare (1.1.1.1), and more. Although you can change your default DNS, ISPs are dumb and lame and sometimes restrict features if you don't use their designated DNS server. When you buy a domain name through a domain name registrar, you'll have to setup some DNS records. Essentially, you'll be putting in your domain name and the IP address that you want it to point to. The registrar then sends a request to whichever first party is in charge of maintaining the global DNS record list to update your domain name's address resolution.

8 HTTPS and Web Certificates

We know that the HTTP protocol is unsecure, and that HTTPS uses TLS/SSL to encrypt packets as they are sent over the network. Using HTTPS is not just as simple as changing a flag though. If you're familiar with how asymmetric end-to-end encryption (E2EE) works, a public key can be shared with any number of users, which can only be decrypted with by computing a mathematical hash using the value of a private key. A hash function is a one-way i.e. non-reversable algorithm that will always produce the same output for a given input. Popular hashing algorithms for encryption include Secure Hash (SHA256) and Message Digest (MD5). Popular encryption algorithms for actually generating the keys are Rivest,Shamir,Adleman (RSA) and Advanced Encryption Standard (AES). In order to use HTTPS we need an SSL certificate. This certificate verifies the authenticity of the web server that hosts the content. As you could imagine, hackers might try to exploit the end user by redirecting their search to their server where they can perform mallicious attacks. This is the infamous "Hackers might be trying to steal your information" message that you get

sometimes. The certificate contains a public key that can be validated by your browser to ensure that the IP address it received from the DNS server was, in fact, your server. It used to be the case that you'd have to pay about $100 annually to get a certificate, but now there are projects like Let's Encrypt and Cloudflare that provide certificates for free! Certificates are obtained from Certificate Authorities (CAs). Let's Encrypt is one such CA started by the EFF, Mozilla, and others. There are tools like certbot that makes fetching these certificates even easier. Certificates must be renewed monthly, but this can also be automated trivially.

9 Static vs Dynamic Websites

10 Static Site Generators

11 Firewalls

A firewall limits the ports that can and can't receive traffic on your machine. The default firewall on most Linux distros is Uncomplicated Firewall (UFW). UFW is very easy to use. It will likely begin by having ports 80 and 443 enabled. We can enable ports for both IPV4 and IPV6, and also for TCP or UDP. As I mentioned before, up until recently the web has been running on TCP, so most of the time we only enable ports on TCP, but you do have the option if UDP is needed.

12 Proxies

A proxy is a server which acts as an intermediary party between two relays. Virtual Private Networks (VPNs) are a good example of a proxy. If you have a VPN (well first of all, you should cancel because they are scams) then your traffic is first being sent to their server, where they then spoof your IP address, and then forward your request to your ISP. One of the big selling points of VPNs is that they are encrypted, but so is 90% of the web through HTTPS... All that you've done by paying for a VPN is brought in another party to potentially snoop on your data, alongside your ISP, but I digress. Proxies can be used for a plethora of things since they are just servers at the end of the day. The Tor browser uses proxies to completely anonymize you by forwarding your request through 3 random relays i.e. proxies on the Tor network before the request exits to the internet (this is definitely what you should be using for privacy instead of VPNs).

A reverse proxy is an application (usually ran as a daemon) that forwards client requests to a back end application, such as your website. The difference between a reverse proxy and a forward proxy is that forward proxies accept connections from computers on a private network and forwards them to the internet, whereas reverse proxies accept requests from the internet and send them to computers on a private network.

13 Apache Server, NGINX, and Python Server

We're now going to look into methods of hosting websites. Note that developers typically host their websites on their loopback address (localhost) prior to release since it is easier to connect to your local machine and modify code on your local machine, not to mention that fact that you don't want to be paying for a web server while you're developing the code. Apache's HTTP server is a classic web server that pretty much used to be the way every website was hosted. In fact, if you run a Linux distro, you probably have a directory /var/www/ which is the root directory for the Apache server. In order to run apache, execute the httpd command (stands for Apache HTTP Daemon) e.g. /usr/bin/httpd (the command should ran using the full path). Now, going to localhost:80 or 127.0.0.1:80 will direct to the default Apache HTTP page. Replacing this page with your own index.html file in /var/www/ will then display that instead.

If you just need a super simple HTTP server quickly and easily on your local machine, python server is, in my opinion, the way to go. If you have Python installed, just run python3 -m http.server and it will launch a new server at 0.0.0.0:8000. Unlike Apache HTTP server which uses /var/www/ as its root directory, python

server will just use the directory that you're currently in. If no index.html page is found at the root directory, python server sets up an FTP server for you!

My personal favorite way of setting up actual websites has to be nginx, mostly due to its easy integration with certbot. NGINX is a reverse proxy that you can install on your web server that will direct requests coming from the internet to your website or web application. NGINX has a bit of a barrier to entry being that you need to be able to write config files for your websites that NGINX is capable of parsing.

14 Routing

Routing is a mechanism for loading pages within a web application that has grown in popularity over the years. For many static websites, routing is a bit redundant, since anchor tags provide us the ability to load pages just fine. Routing comes in various forms, but generally, the way that it works is by mapping the endpoint of the URI to a function on the server. The function will usually (though not always) fetch the appropriate page. An endpoint is simply the resource located at the URI's path. For instance, pretend that we navigate to https://www.example.org/get/the/endpoint. In this case, endpoint may not be an actual file on our server, but rather, a function that will be invoked to return endpoint.html or run some other task completely. This method of routing is known as path-based routing, as opposed to host-based, which is the conventional way of simply returning the thing that the user requested. Another less common type of routing is header-based routing, in which we route depending upon the headers that are present in the HTTP request. Typically, we use custom headers to achieve this. Historically, custom HTTP headers used to start with an 'X' prefix, but this was deprecated in 2012, so now you just have to lookup the header to see if it's proprietary or not.

15 Dynamic Websites

15.1 Server-Side Applications

15.2 Client-Side Applications

15.3 Page Rehydration

16 JavaScript Frameworks

16.1 React

React is arguably the most popular framework on the market in current year. This is mostly in part thanks to its backing by Facebook, and its more mature lifetime (frameworks such as Angular, Vue JS, and Svelte are all significantly more recent). One thing that I think confuses new web developers learning React is its use of JSX. JSX is truly a Frankenstein mishmash of JavaScript and HTML, which makes it sometimes difficult to distinguish. For instance, it is possible using JSX to assign an HTML element to a JavaScript variable.

```
const element = <h1>This line of code uses JSX</h1>
```

What React fundamentally allows us to do is create custom JSX elements/components which look like HTML tags and accept attributes in the same manner. This powerful heirarchical pattern allows us to build components of off other pre-existing components, which makes development quicker and less repetitive. Practically speaking, components are created by returning a block of JSX code from a function. A new tag is then generated based off the name of the function. React components must use Pascal casing, which

distinguishes them from regular functions. For example, we can create a custom button component with the following code:

```
const CustomButton = (): JSX.Element => {
    const sayHello = () => {
        alert('Hello, World');
    }

    return (
        <div>
            <button
                style={ color: 'red' }
                onClick={ sayHello }
            >
                My Custom Button
            </button>
        </div>
    );
};

export default CustomButton;
```

Then elsewhere, we can import this component and use it like so (note that App() is the entry point for React):

```
import CustomButton from 'path/to/file/CustomButton';

function App(): JSX.Element => {
    ...
    <CustomButton />
    ...
};

export default App;
```

We can escape JSX and insert JavaScript code using curly braces. This is what we did for the style and onClick attributes in the former example. Likewise, any comments must first be escaped like so:

```
{ /* A comment in JSX */ }
```

React uses hydration to update individual components, rather than reloading the entire DOM each time the state of a component changes. This is, naturally, much faster than attempting to manipulate individual elements in the DOM, which is what something like jQuery does. Once again, looking at the practical application in terms of how this is achieved, React uses something called hooks (essentially a glorified name for callback functions) to update Components when the state is changed. There are many kinds of hooks, which we'll look at shortly, but the relevant kind that we're interested in for updating components

on state change are state hooks. You can tell that a function is a hook in React if it begins with the word 'use'. State hooks are created using the useState() function. What this does is return a list with exactly two items. The first item it returns will be the variable that will maintain state, and the second item will be a callback function that can be used to update the state of this variable. The idea is to keep the state variable as immutable i.e. we should not update the state variable directly, but rather indirectly, via the callback. Typically, we name the state variable some relevant name, and we name the callback the same thing, but prepended with 'set', akin to a setter in OOP languages. One final note before I completely lose you: the useState() hook can take a parameter (in TS this would be of type any) and assign that value as the initial state for the state variable. Let's look at this in code:

```
function App(): JSX.Element {
    const [counter, setCounter] = useState(3);
    return (
        <div>
            <h1>{ counter }</h1>
            <button onClick={ () => { setCounter(counter + 1); } }>
                Increment counter
            </button>
        </div>
    );
}

export default App;
```

17 Managing Dependencies

JavaScript code tends to have a lot of dependencies. Managing these dependencies can be quite a hassle if you're trying to do it by yourself. You will definitely want a package manager to be able to install, uninstall, update, and manage your project's dependencies. The two primary contenders for handling this are the Node Package Manager (npm) and yarnpkg (usually just called yarn).

17.1 npm

npm is the official package manager for JavaScript, and is probably what I would recommend if you're just starting out. npm is a pretty straightforward package manager, though there are a couple of things to note before you go out and start installing a bunch of packages. First of all, there is a ton of crap on npm since anyone can easily upload their libraries to the upstream repo. npm will mark vulnerable packages as such, but this will only occur after the vulnerabilities have been reported and confirmed.

To find a package by keyword, use npm search [search terms]. If a term begins with a forward slash, it shall be interpreted as a regex pattern. When it comes to installing packages, you have a few options that you might want to consider. Before I get carried away, note that you can run npm help <subcommand> to bring up a man page containing additional information about that subcommand. For install, the most important option is the -g or –global flag, which