

# Server Farm to Table

(or, how the Internet works)

Jenna Zeigen  
@zeigenvector

hey, sup?



# Server Farm to Table

(or, how the Internet works)



nm, u?

Jenna Zeigen  
@zeigenvector

So, what **\*does\*** happen when you type [www.google.com](http://www.google.com) into the browser?



Jenna Zeigen • @zeigenvector

The OSI model is a really good way of thinking about processes like these. It details seven layers, from Layer 1 which talks about physical bits in the computer, through Layer 7, which is the full application layer, so things like the browser. This talk is going to start at Layer 4, which is transport. We won't be talking about anything lower than that, so no networking, though I'm sure there's plenty of stuff all over the Internet if you want to learn more about those lower layers.

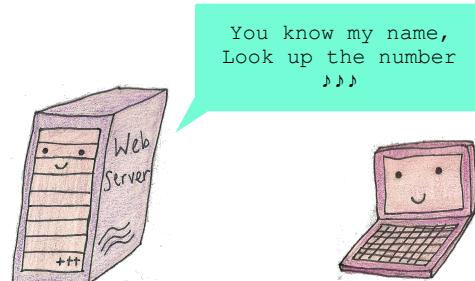
# Overview

1. IP Address Lookup
2. Opening a socket
3. Security Stuff
4. HTTP Request
5. Server Things
6. HTTP Response
7. Parsing
8. Rendering

Jenna Zeigen • @zeigenvector

I'm going to go through all of these and talk about optimizations along the way!

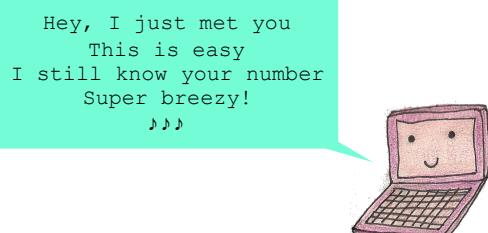
## IP Address Lookup



Jenna Zeigen • @zeigenvector

To do anything, we need to map the human readable URL "[www.google.com](http://www.google.com)" to the address of a computer to "talk to."

## IP Address Lookup



Jenna Zeigen • @zeigenvector

First, the browser checks its cache.

## IP Address Lookup

Is this name special?  
So very special?  
♪ ♪ ♪



## IP Address Lookup

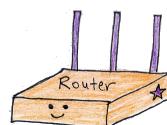
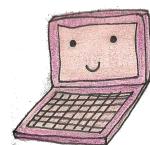
Nope!



## IP Address Lookup

Do you have  
a www.google.com?

You weren't asking  
me, but I know  
where that is!  
173.194.46.84!



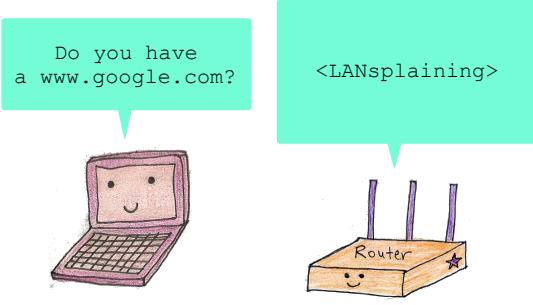
If not found, the browser calls the `gethostbyname` library function, the contents of which vary by OS, to do the lookup. This function checks if the hostname is listed in the local hosts file, the location of which also varies by OS. On Macs, it's at `/etc/hosts`. This is how your computer knows what `localhost` should resolve to. If you wanna check it out, you can type `cat /etc/hosts` into your terminal and see what's in there!

If `gethostbyname` does not have it cached and can't find it in the hosts file, then it makes a request to the DNS server configured in the network stack. This is typically the ISP's caching DNS server, but you could configure your computer to use any DNS server, i.e. Google's (8.8.8.8) if you wanted...

## IP Address Lookup

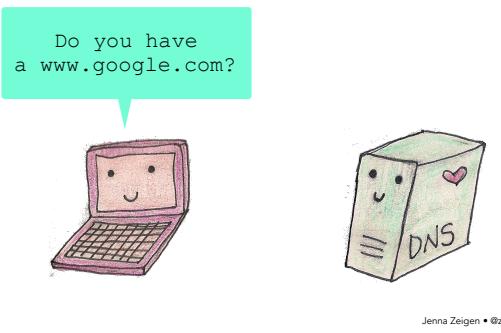
The browser then kicks off a request to get the IP address for `www.google.com`, which goes through the local router, which also checks its cache.

## IP Address Lookup



(Big thanks to @jedschmidt for this witty commentary)

## IP Address Lookup

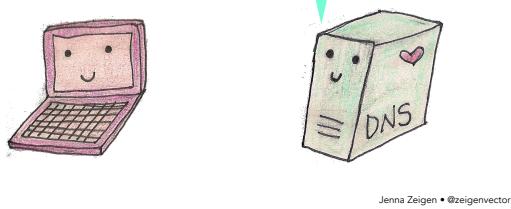


If the local router doesn't have anything cached, the browser ends up hitting the local ISP DNS server. This process uses UDP (User Datagram Protocol), or TCP if the request is too large.

UDP is kind of like the smallest bit of network you can use. It's great for unidirectional transactions, but delivery is not guaranteed in any order, or guaranteed at all. It's kind of like if you were to throw a bunch of balls at someone and hope that they catch them.

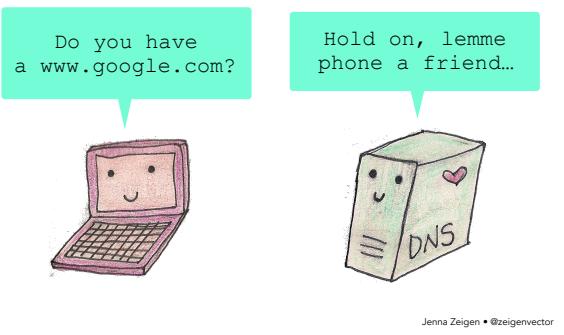
## IP Address Lookup

Got a long list of these numbers  
But if it's not in my "brain"  
I'll ask some servers  
And we'll find it by name  
♪♪♪



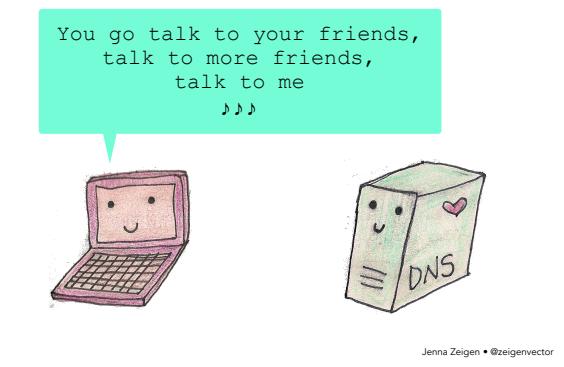
Upon receiving the request, the DNS server first checks its cache.

## IP Address Lookup



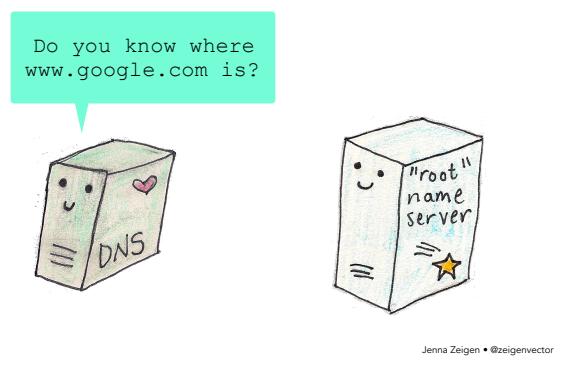
If it doesn't have the address in its cache, it recursively looks up the address by asking some more servers...

## IP Address Lookup



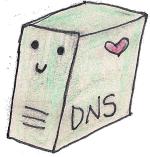
(Meanwhile, the client has to wait a while for the answer...)

## IP Address Lookup

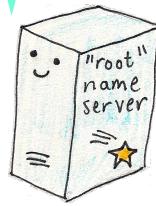


## IP Address Lookup

Do you know where  
www.google.com is?



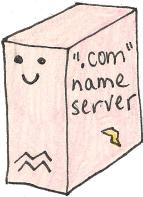
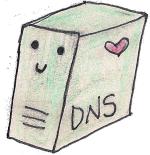
No, but maybe ask  
.com?



Jenna Zeigen • @zeigenvector

## IP Address Lookup

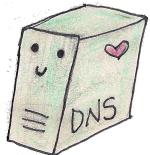
Do you know where  
www.google.com is?



Jenna Zeigen • @zeigenvector

## IP Address Lookup

Do you know where  
www.google.com is?



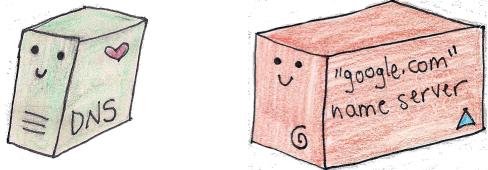
No, but maybe ask  
google.com?



Jenna Zeigen • @zeigenvector

## IP Address Lookup

Do you know where  
www.google.com is?

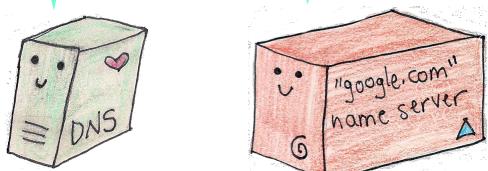


Jenna Zeigen • @zeigenvector

## IP Address Lookup

Do you know where  
www.google.com is?

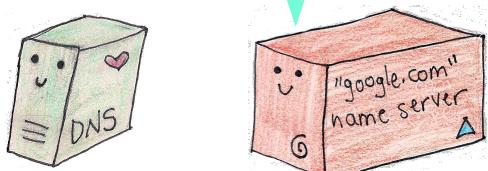
Yep!  
173.194.46.84!



Jenna Zeigen • @zeigenvector

## IP Address Lookup

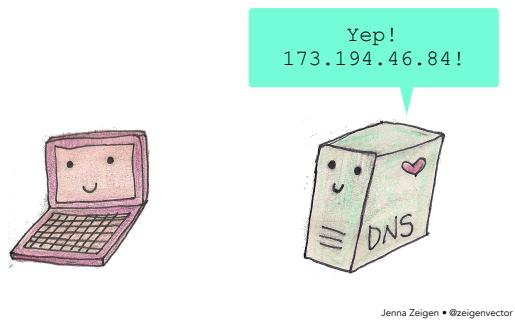
Say you'll remember me  
Or at least this IP address  
So we don't have to relive this mess  
♪♪♪



Jenna Zeigen • @zeigenvector

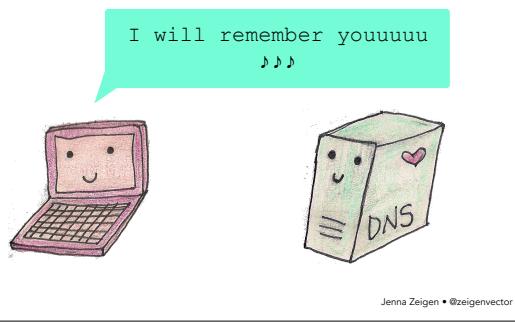
When the name server comes back to the DNS server with the IP address answer, it will also come back with a TTL (time to live), an amount of time that the DNS server should cache the IP address. The longer the TTL, the faster this process, but then it's harder to change things. Historically, this TTL was 24 hours, which is why it could take up to 24 hours for the URL corresponding to an IP to change.

## IP Address Lookup



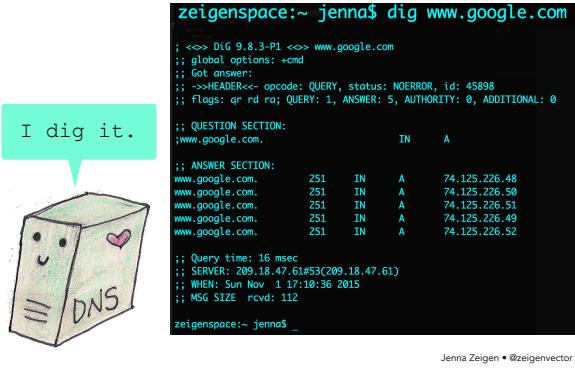
The DNS server then responds to the client with the found IP address.

## IP Address Lookup



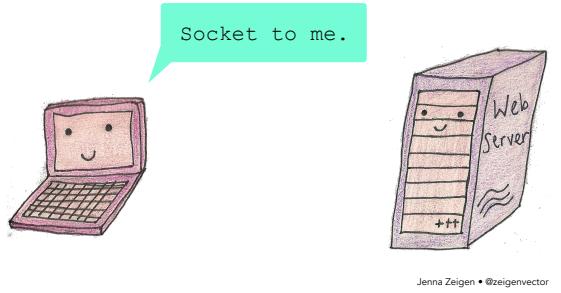
The client then caches the IP address for next time. Some browsers also have their own DNS cache.

## IP Address Lookup



If you want to check out this process hands-on, you can use the `dig` command in your terminal.

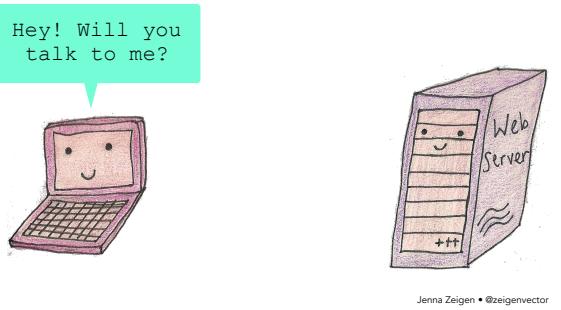
## Opening a socket



Once the client has the IP address, it takes it and the port number from the URL, and makes a call to the system library function named `socket` and requests a TCP socket stream. We don't usually see port numbers on the end of URLs, but the HTTP protocol defaults to Port 80, and HTTPS to Port 443.

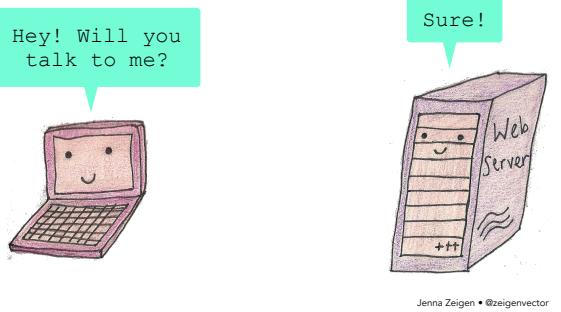
TCP stands for "Transmission Control Protocol." Unlike UDP, its job is to deliver a reliable, ordered, and error checked stream of packets across the network. It's more like a game of catch, to continue the ball metaphor. In fact, it's often called the "TCP handshake". TCP is the protocol that underlies HTTP, TLS, FTP, email, and SSH.

## Opening a socket

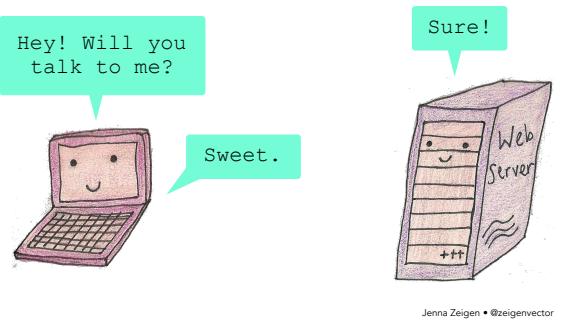


The client then establishes TCP connection(s) with the server. It's kinda like this...

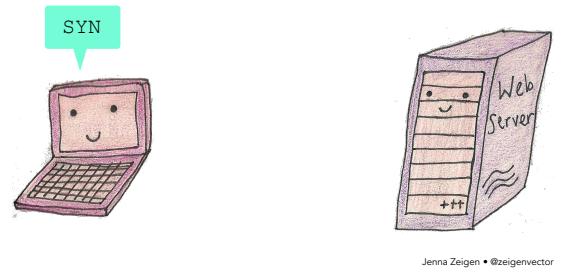
## Opening a socket



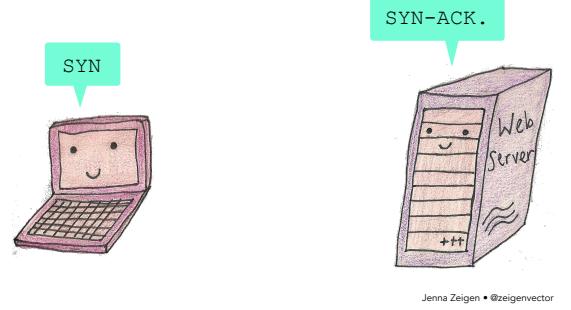
## Opening a socket



## Opening a socket



## Opening a socket

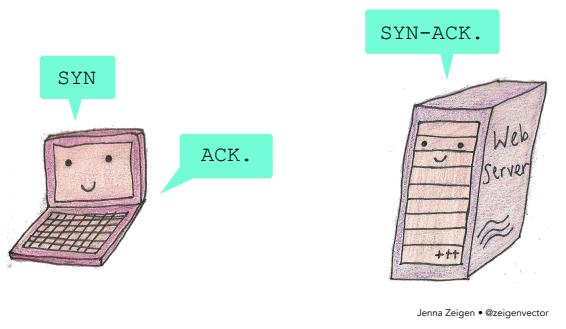


Well, more specifically like this...

1. The client sends a SYN packet with a randomly-selected sequence number "x"

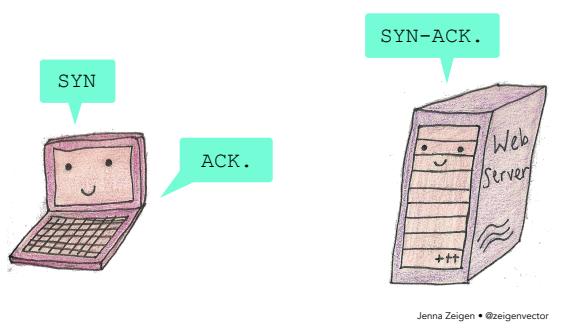
2. The server replies with a SYN-ACK packet with an acknowledgement number set to "x+1" and a different randomly-selected sequence number "y"

## Opening a socket



3. The client responds with an ACK packet, with the sequence number set to the received acknowledgement number " $x+1$ ", and a new acknowledgement number set to " $y+1$ "

## Opening a socket



To prepare for the likely event that the client is going to need more things from the server, more than one connection will be opened.

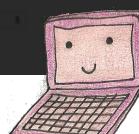
TCP also has congestion control—requests start small and ramp up in size to make sure the network can support the requests. This is more of a relic from the past when networks weren't as reliable or fast.

In addition, TCP is where much the latency in this entire process comes from, as it's constrained by the speed of light. This is why it is advantageous to server assets from a server as close to your user as possible.

## Opening a socket

```
zeigenspace:~ jenna$ sudo tcpdump -c3 host www.google.com
tcpdump: data link type PKTAP
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ptkap, link-type PKTAP (Packet Tap), capture size 65535 bytes
01:05:42.209216 IP 192.168.0.23.52293 > lga25s40-in-f196.1e100.net.https: UDP, length 190
01:05:42.249138 IP lga25s40-in-f196.1e100.net.https > 192.168.0.23.52293: UDP, length 35
01:05:42.259526 IP lga25s40-in-f196.1e100.net.https > 192.168.0.23.52293: UDP, length 221

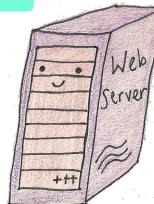
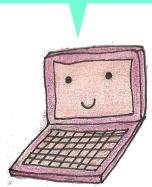
3 packets captured
16 packets received by filter
0 packets dropped by kernel
zeigenspace:~ jenna$
```



If you want to check out TCP for yourself, you can use `tcpdump` from your terminal. `tcpdump` will show you all the network traffic, which is why you see UDP packets here. If you prefer a GUI option, you can also use [Wireshark](#).

## Security Stuff

After all this goes down,  
I know that we'll be safe and sound  
♪♪♪

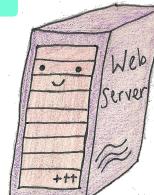


Jenna Zeigen • @zeigenvector

If the request is over HTTPS, we need another step to ensure the security of the transaction. HTTPS uses TLS, or Transport Layer Security. TLS's predecessor was Secure Sockets Layer (SSL).

## Security Stuff

ClientHello! It's me,  
I wish to make this exchange safe  
And hope that you'll agree  
♪♪♪

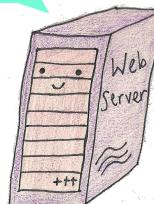


Jenna Zeigen • @zeigenvector

To kick off the process, the client computer sends a ClientHello message to the server with its TLS version, list of cipher algorithms and compression methods available.

## Security Stuff

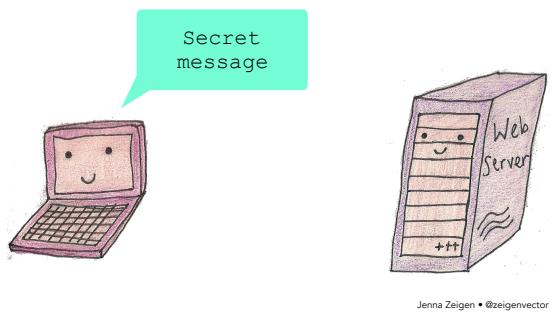
ServerHello from the other side!  
Here's a certificate and public key!  
Let's try!  
♪♪♪



Jenna Zeigen • @zeigenvector

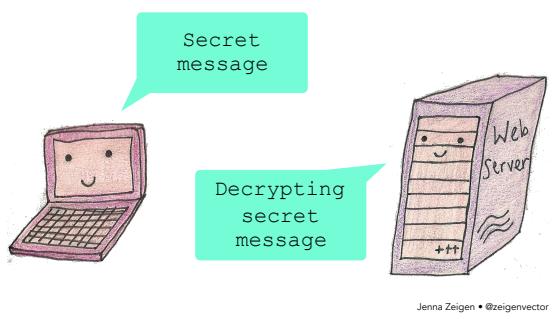
The server replies with a ServerHello message to the client with the TLS version, selected cipher, selected compression methods and the server's public certificate. The certificate contains a public key that will be used by the client to encrypt the rest of the handshake until a symmetric key can be agreed upon.

## Security Stuff



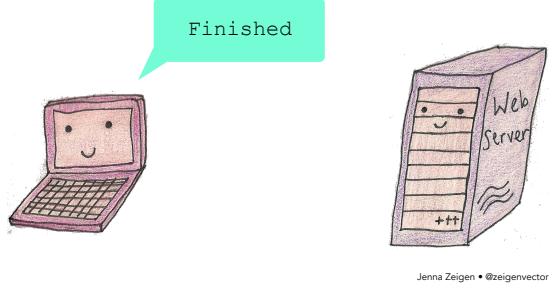
The client verifies the server digital certificate. If trust can be established, the client generates a string of pseudo-random bytes and encrypts this with the server's public key.

## Security Stuff



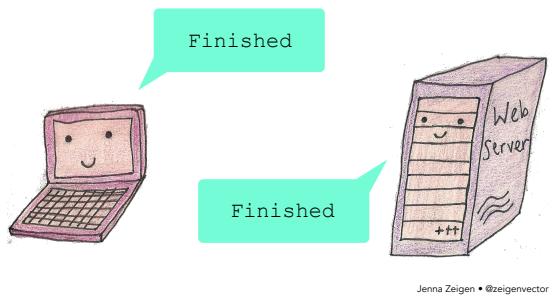
The server then decrypts the message and uses it to create its own version of the symmetric key.

## Security Stuff



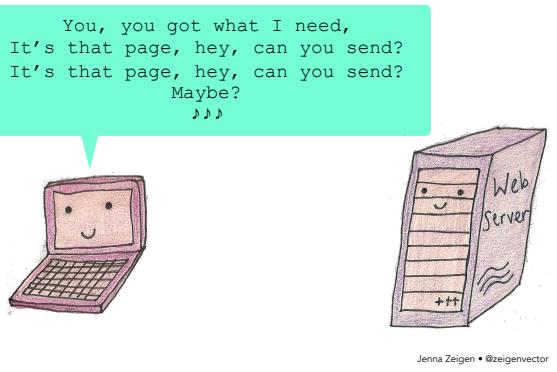
The client sends a Finished message to the server, encrypting a hash of the transmission up to this point with the symmetric key.

## Security Stuff



The server generates its own hash, and then decrypts the client-sent hash to verify that it matches. If it does, it sends its own Finished message to the client, also encrypted with the symmetric key.

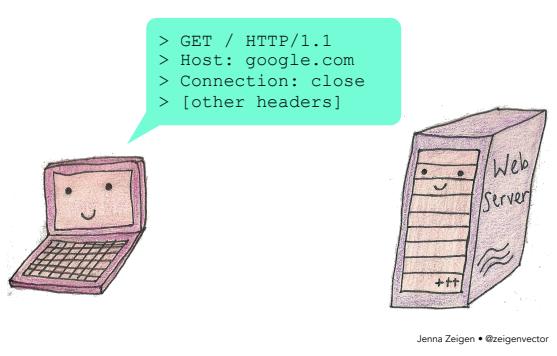
## HTTP Request



With all this done, the client is ready to send an HTTP request to the server. HTTP stands for "HyperText Transfer Protocol." Unless the client and server negotiate using a more advanced version of HTTP, they're going to use HTTP/1.1.

HTTP/1.1 is stateless, uses a request/response model, and is a plain-text protocol.

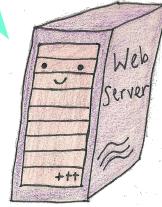
## HTTP Request



The client will send a request like this, containing the request and headers, followed by a single blank newline to the server indicating that the content of the request is done.

## Server Things...

Look inside  
It's where my daemon hides  
♪♪♪

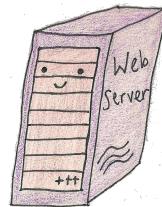


Jenna Zeigen • @zeigenvector

The server then processes the request. The HTTPD (HTTP Daemon) server is the one handling the requests and responses on the server side. Some popular HTTPD servers are Apache and nginx.

## Server Things...

I got racks on  
racks on racks  
♪♪♪

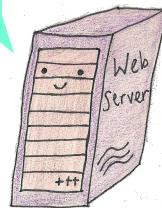


Jenna Zeigen • @zeigenvector

The server is going to do a bunch of "server stuff" including load balancing, routing, and database calls.

## Server Things...

Whatcha, whatcha,  
whatcha want?  
(Whatcha want?)  
♪♪♪



Jenna Zeigen • @zeigenvector

The server breaks down the request to the following parameters:

1. HTTP Request Method (either GET, POST, PUT, etc.). In this case, with a URL entered directly into the address bar, this will be GET.
2. Domain. In this case the domain is "[google.com](http://google.com)."
3. Requested path/page. In this case, the path is "/" because no specific path or page was requested (/ is the default, root path, just like in the Unix file system).

## HTTP Response

< HTTP/1.1 304 Not Modified  
[other headers]



If you've been to this site before, and the HTTP headers sent by the web browser included sufficient information for the server to determine if the version of the file cached by the web browser has been unmodified since the last retrieval, it may instead respond with something like this.

This is so much less work for the client and server at this point, but is the result of author work, making sure that the proper mechanisms are in place for the required data to be transmitted.

## HTTP Response

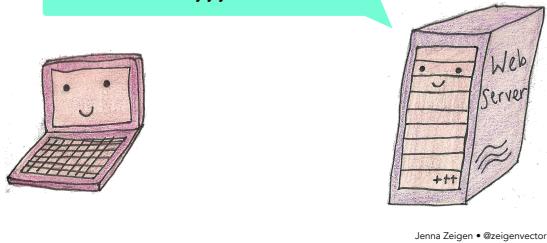
Sorry, we had to change to that code  
Updated that file and then changed its number  
I guess you need the new stuff though  
Now it's just some page that you used to know  
♪ ♪ ♪



For example, having ETags in the header will allow the server to determine if the cached version of a page is still valid. The first time the server sends the page, it will send an ETag, a unique identifier for the version of the page, in the header of the response. The client will then cache this. The next time the client needs this page, it will get the ETag from its cached version and send it to the server. If the ETags match, the server will send the 304 Not Modified status code.

## HTTP Response

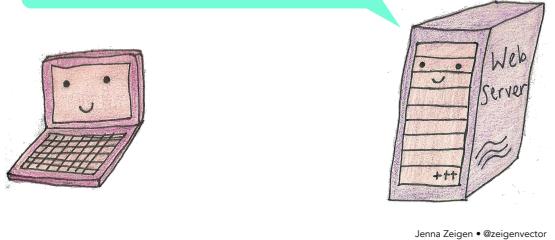
All this HTML  
I will send to you  
All this HTML  
You'll know what to do  
♪ ♪ ♪



If they don't match, then there's much more to be done. The server has to then send over a much bigger payload.

## HTTP Response

```
< HTTP/1.1 200 OK
[other headers]
<
<!DOCTYPE html><html>...</html>
```

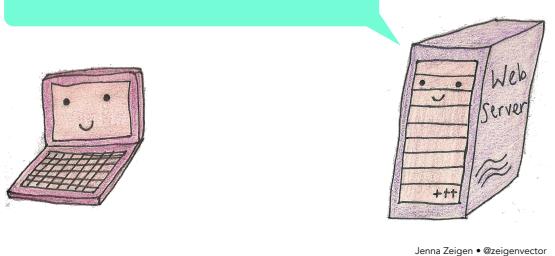


Jenna Zeigen • @zeigenvector

An otherwise successful request and smooth servering process will prompt the server to send a response like this, with a 200 OK status code, other headers, a single blank newline, and then the payload, which in this case is the entire HTML document.

## HTTP Response

```
It didn't work, work,
work, work, work, work!
♪ ♪ ♪
```



Jenna Zeigen • @zeigenvector

If something went wrong, you could also get less-happy status codes. The 400-series represents a client error like 403 Forbidden or 404 Not Found. The 500-series represents server errors, such as the 500 Internal Server Error.

## HTTP Response



If you want to investigate what's going on in an HTTP request, you can use `curl` from your terminal.

## HTTP Response

```
zeigenspace:~ jenna$ curl www.google.com  
<!doctype html><html itemscope="" itemtype="http://
```

My curl, my curl, my curl  
Talkin' 'bout my curl.

My curl!  
♪♪♪



## HTTP Response

Shrink it down,  
gzip it  
(Don't reverse it...)  
♪♪♪



Jenna Zeigen • @zeigenvector

## HTTP Response

These are my compressions  
♪♪♪



Jenna Zeigen • @zeigenvector

With HTTP, the size of the request directly correlates to the time it takes to finish the request. Because of this, compression helps a lot to get your page to the user as fast as possible. Gzip is most popular compression method for these web things. Gzip works by replacing repeated substrings with references to where the decompressor can find that substring, and this is great for HTML (think of how many times div appears in an HTML document! In fact, gzipping generally reduces the response size by about 70%.

And if you're worried your users don't have this fancy compression method enabled in their browsers, the server and client talk about using gzip beforehand to make sure it's safe. And about 90% of today's Internet traffic travels through browsers that claim to support gzip anyway.

(Thanks to @jamessocol for this one)

## HTML Parsing

```
1 <!doctype html>
2 <html>
3   <head>
4     <script type="text/javascript" src="library.js"></script>
5     <link rel="stylesheet" type="text/css" href="styles.css">
6     <script type="text/javascript" src="website.js"></script>
7   </head>
8   <body>
9     <div>Let's pretend this is all of google</div>
10   </body>
11 </html>
```



We did everything right,  
Bytes are on the client side  
♪ ♪ ♪

Jenna Zeigen • @zeigenvector

## HTML Parsing

```
1 <!doctype html>
2 <html>
3   <head>
4     <script type="text/javascript" src="library.js"></script>
5     <link rel="stylesheet" type="text/css" href="styles.css">
6     <script type="text/javascript" src="website.js"></script>
7   </head>
8   <body>
9     <div>Let's pretend this is all of google</div>
10   </body>
11 </html>
```



Are you listening? (Whoa-oh-oh-oh-oh)  
Please come back. (Whoa-oh-oh-oh-oh)  
I'll tell you what do I need  
I'll tell you what do I need  
Whoa-oh, whoa-oh  
♪ ♪ ♪

Jenna Zeigen • @zeigenvector

## HTML Parsing

```
1 <!doctype html>
2 <html>
3   <head>
4     <script type="text/javascript" src="library.js"></script>
5     <link rel="stylesheet" type="text/css" href="styles.css">
6     <script type="text/javascript" src="website.js"></script>
7   </head>
8   <body>
9     <div>Let's pretend this is all of google</div>
10   </body>
11 </html>
```



I begin to parse,  
To split the text apart  
Break it down into sections  
Tokens into selectors  
♪ ♪ ♪

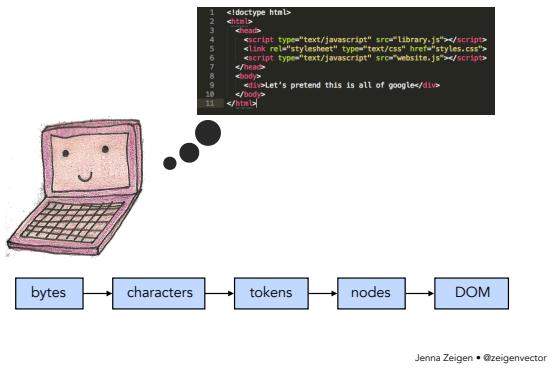
Jenna Zeigen • @zeigenvector

As soon as the browser starts receiving bytes, it starts parsing it. It usually receives it in 8kB chunks, and feeds those bytes right into the parser.

Before the formal parsing for rendering, the bytes get passed through a speculative parser, or look-ahead pre-parser. This parser looks ahead for external resources like JS, CSS, and image files so it can start getting them from the server immediately. If it finds them, it'll use the extra TCP requests, if they're from the same host. Otherwise, the client has to do the whole DNS dance again. This optimization wasn't always in the stack, but when released, improved page load performance by about 20%.

Meanwhile, the bytes are fed into the primary parser.

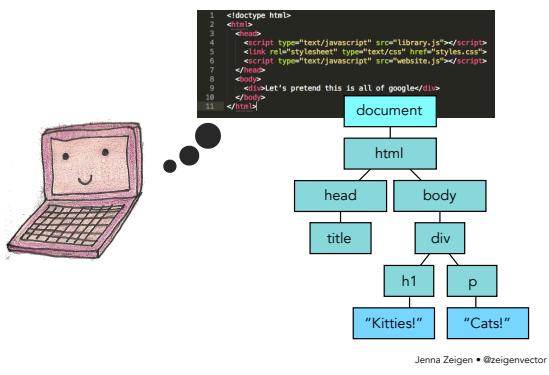
## HTML Parsing



The parsing algorithm is described in detail by the HTML5 specification. It goes something like this:

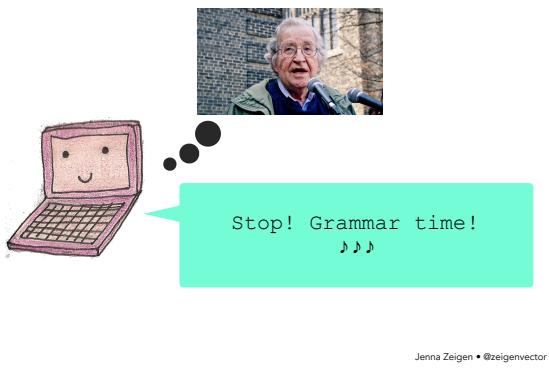
1. The browser reads the raw bytes of the HTML off the disk or network and translates them to individual characters based on specified encoding of the file, e.g. UTF-8.
2. The characters get broken up into tags, e.g. html, head, and div.
3. The tags then then get made into nodes.
4. The nodes are then arranged into the DOM tree.

## HTML Parsing



After this process, you end up with a DOM (Document Object Model) tree made from the HTML document. It has an almost one-to-one relationship with the markup.

## HTML Parsing



Most programming languages have a vocabulary described using regular expressions and a syntax described by a context-free grammar. A context-free grammar means there is a set of production rules that describe all possible strings in the language. Production rules are simple replacements that can be applied regardless of context.

(Photo credit: <https://www.flickr.com/photos/andrewrusk/5599576288>)

## HTML Parsing

```
1 <!doctype html>
2 <html>
3   <head>
4     <script type="text/javascript" src="library.js"></script>
5     <link rel="stylesheet" type="text/css" href="styles.css">
6     <script type="text/javascript" src="website.js"></script>
7   </head>
8   <body>
9     <div>Let's pretend this is all of google</div>
10   </body>
11 </html>
```



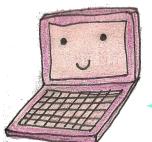
Whatever, wherever  
I'm gonna make it render!  
♪ ♪ ♪

Jenna Zeigen • @zeigenvector

HTML isn't defined by a context-free grammar and therefore cannot be parsed by a "regular" parser. This is because HTML is designed to be more forgiving of human error—the spec requires the parser to be 100% fault-tolerant. This is why you never see an HTML syntax error. It will always render something (but sometimes it's not *\*exactly\** what you wanted...)

## HTML Parsing

```
1 <!doctype html>
2 <html>
3   <head>
4     <script type="text/javascript" src="library.js"></script>
5     <link rel="stylesheet" type="text/css" href="styles.css">
6     <script type="text/javascript" src="website.js"></script>
7   </head>
8   <body>
9     <div>Let's pretend this is all of google</div>
10   </body>
11 </html>
```



All the right tags  
In all the right places  
♪ ♪ ♪

Jenna Zeigen • @zeigenvector

Thus, a lot of the parser's code is for fixing the HTML author mistakes, like invalid tags, unclosed tags, incorrect nesting.

## HTML Parsing

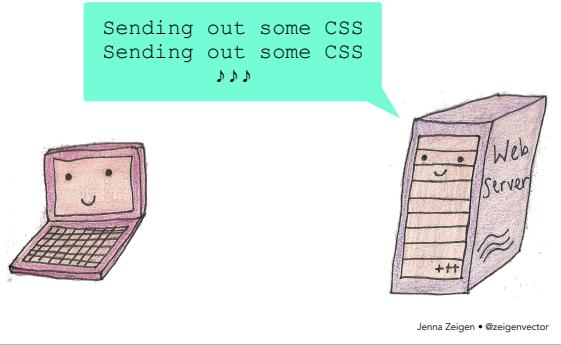
Yeah, my momma she told me don't worry  
about your size  
She says I'm this big 'cause sometimes  
humans just aren't so bright  
♪ ♪ ♪



Jenna Zeigen • @zeigenvector

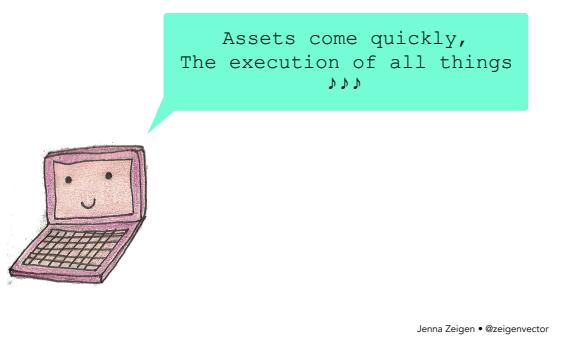
This means the parser will never error, but it also means that browser makers have to create custom parsers. Browsers are like a million lines of C++, and a big chunk is related to this parser.

## HTML Parsing



As the parser is parsing, it runs into inlined assets like images, scripts, and stylesheets. The client must then request each asset and wait for the server to send it back. Hopefully the speculative parser has already started the fetching process!

## HTML Parsing



Once the browser has the asset, it then must parse and execute it before anything continues.

## HTML Parsing



The big hiccup here is that browsers have imposed limits on how many parallel downloads you can have from the same domain. It used to be 2, but now that limit is anywhere from 6 to 13 in modern browsers, depending on the browser. This is a huge bottleneck, but many of the optimizations we hear about stem from trying to get around this. One thing you can do here is to serve your images from multiple hostnames, you can get around the same hostname parallel download limit.

## HTML Parsing

Pull it down, I'm yelling timber  
Unless it moves or gets enhanced  
This data, I'm told to remember  
Until it tells me to forget  
♪♪♪



Jenna Zeigen • @zeigenvector

## HTML Parsing

Come together, right now  
Up the speed  
♪♪♪



Jenna Zeigen • @zeigenvector

## HTML Parsing

Yeah, it's always better  
when they're together  
♪♪♪



Jenna Zeigen • @zeigenvector

Caching also comes into play here, because TTL in this case is determined by the headers sent with the asset, such as a far-future Expires header. This helps because the file does not need to be downloaded again, therefore not contributing to the download limit.

Another popular optimization is to combine your files together. This includes concatenating JavaScript and CSS files together using a tool like webpack, Gulp, or even the asset pipeline. However, this might come back to bite you with caching because overtime you change just one file in your bundle, the cache gets invalidated. Because of this, some developers have two bundles—one for application code that is likely to change often and one for vendor code that is less likely to change.

Another thing you can do is make CSS sprites or icon fonts so there aren't so many small images each with their own requests.

## HTML Parsing

Got to keep you inlined  
To make this page load streamlined  
♪♪♪



Jenna Zeigen • @zeigenvector

## HTML Parsing

Harder, Better,  
Faster, Smaller  
♪♪♪



Jenna Zeigen • @zeigenvector

## HTML Parsing

Got some files to send over  
And their length's insane  
Remove the whitespace, baby  
It'll up your game  
♪♪♪



Jenna Zeigen • @zeigenvector

Some developers even inline their images as data:URIs.

Still, with referenced assets the size of the request affects the time it takes to download, so you want your requested files to be as small as possible. This takes some effort on the part of the developer.

One thing you can do is minify all your HTML, CSS, and JavaScript. While it's good to write your code for humans with whitespace and comments, computers don't care about all of this. Minification gets rid of all the stuff we put in code for humans.

## HTML Parsing



All the small things  
What speed this brings!  
»»»

Jenna Zeigen • @zeigenvector

## HTML Parsing



So I don't take the fall  
Of a document.write call  
When I see you, everything stops  
Probably never put you on top  
»»»

```
<head>
<script src="library.js"></script>
<link href="styles.css">
<script src="website.js"></script>
</head>
<body></body>
```

Jenna Zeigen • @zeigenvector

## HTML Parsing



Scripts go at the bottom  
Not up here  
Scripts go at the bottom  
So the whole page freakin' here  
»»»

Jenna Zeigen • @zeigenvector

Optimizing images is also important. JPEGs can be huge, especially for mobile devices, and SVGs especially have extra information that can be removed if run through an optimizer such as [svgo](#).

As the parser is doing its parsing and it runs into a JavaScript include, further parsing stops. This is because JavaScript has the ability to manipulate the DOM, using `document.write`, and any HTML that gets added must be fed back into the main parser. The parsing of the document halts until the script has been downloaded, if necessary, and executed. While a script is downloading, the browser won't start any other downloads, even on different hostnames.

This explains the common optimization of putting scripts at the bottom of the page, right before `</body>`. Thus, when this blocking inevitably occurs, all the good things have already happened and you're well on your way to a render!

## HTML Parsing

We can't stop...  
♪♪♪



```
<head>
<script defer src="library.js"></script>
<link href="styles.css">
<script src="website.js"></script>
</head>
<body></body>
```

Jenna Zeigen • @zeigenvector

## HTML Parsing

And we won't stop...
 ♪♪♪



```
<head>
<script async src="library.js"></script>
<link href="styles.css">
<script src="website.js"></script>
</head>
<body></body>
```

Jenna Zeigen • @zeigenvector

## HTML Parsing

Stop right now
 Thank you very much
 I need all the CSS downloaded and such
 ♪♪♪



```
<head>
<script src="library.js"></script>
<link href="styles.css">
<script src="website.js"></script>
</head>
<body></body>
```

Jenna Zeigen • @zeigenvector

There are also spec-approved ways to get around this blocking too. Authors can add a "defer" attribute to a script tag, in which case it will not halt document parsing and will execute only after the document is parsed.

HTML5 adds an another option, to mark the script as asynchronous with the `async` attribute so it will be parsed and executed by a different thread, in parallel. With this, you're kinda promising you won't call `document.write`, or you're okay with it getting thrown out the window. (that was an attempt at a DOM pun...)

CSS also does its own kind of blocking. It might seem that since CSS doesn't change the DOM tree, there's no reason to wait for it and stop the document parsing. There is an issue, though, of scripts asking for style information during the document parsing stage. If the styles aren't loaded and parsed yet, the scripts will get wrong answers and apparently this has caused lots of problems. It seems to be an edge case but is quite common.

Different browsers have different ways of dealing with this blocking though. Firefox blocks all scripts when there is a stylesheet that is still being loaded and parsed. WebKit blocks scripts only when they try to access certain style properties that may be affected by unloaded stylesheets.

## CSS Parsing

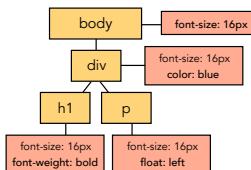
This parser's young and wild and  
context-free  
♪ ♪ ♪



Jenna Zeigen • @zeigenvector

## CSS Parsing

'Cause we know how to style  
We know how to style  
♪ ♪ ♪



Jenna Zeigen • @zeigenvector

## HTML Parsing

Keep on parsing  
'Til the page ends  
♪ ♪ ♪



```
<head>
<script src="library.js"></script>
<link href="styles.css" />
<script src="website.js"></script>
</head>
<body></body>
```

Jenna Zeigen • @zeigenvector

Just like with HTML, the browser needs to convert the CSS file into something it can work with. Instead of making a DOM tree though, it makes a CSSOM tree! The CSS parser, unlike the HTML parser, is context-free, though the steps in the process are largely the same.

The tree data structure is well-suited for CSS because of the recursive nature of style application, the “cascade.” It should be noted that the CSSOM tree only includes overrides to the browser’s default stylesheet, so things that are included in the application’s stylesheet and inlined in the HTML.

So, once the CSS is done, the JavaScript can definitely continue to load.

## HTML Parsing

Keep on parsing  
'Til the page ends  
♪ ♪ ♪



```
<head>
  <script src="library.js"></script>✓
  <link href="styles.css">✓
  <script src="website.js"></script>
</head>
<body></body>
```

Jenna Zeigen • @zeigenvector

Successive assets then can be downloaded and parsed.

## JS Parsing

Scripty's now an abstract syntax tree
Thanks, smart parser (smart parser!)
The next step is the baseline
compiler
It made bytecode (it made bytecode!)
Next thing you know
Scripty got low-low-low-low-low-low
low-low
♪ ♪ ♪



Jenna Zeigen • @zeigenvector

JavaScript is also context-free and can use a regular parser, but browsers these days make it more complicated in order to optimize the process. V8, the JavaScript engine in Chrome, uses two parsers—eager and lazy—to eventually create an abstract syntax tree (AST) and scope structure. The eager parser is the full parser used to parse functions for compilation. It builds an AST, as well as scope structures and finds all syntax errors. The lazy parser is also called the pre-parser. It skips over functions that shouldn't be compiled yet. It doesn't build an AST. It does build scopes, but doesn't process them fully. It also only finds a limited set of errors.

The AST and scope structures get fed into a baseline compiler and turned into low-level code. In the case of V8, a bytecode generator walks the AST and scope structure and turns it into

## JS Parsing

Interpreted by Ignition
Your code coming to fruition
♪ ♪ ♪



Jenna Zeigen • @zeigenvector

The low-level code then gets executed. In V8, the bytecode is then executed by the interpreter, called Ignition in V8

## JS Parsing



Who can say where that byte goes  
Turbofan does, at runtime  
And who can say if your code flows  
Turbofan knows, just-in-time  
♪ ♪ ♪

Jenna Zeigen • @zeigenvector

The bytecode also gets fed to the optimizing compiler, which recompile's functions that get used a lot and are worth speeding up. In V8, there is only one optimizing compiler, called Turbofan. Because JavaScript has a just-in-time (JIT) compiler, the code gets compiled at runtime and information used from initial runs is fed back to the compiler and is used to speed up the function with the assumption it will continue to handle cases similar to what it has already seen

## HTML Parsing



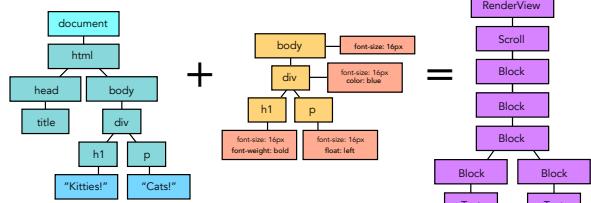
Closing tag...  
♪ ♪ ♪

```
<head>
  <script src="library.js"></script>✓
  <link href="styles.css">✓
  <script src="website.js"></script>✓
</head>
<body></body>
```

Jenna Zeigen • @zeigenvector

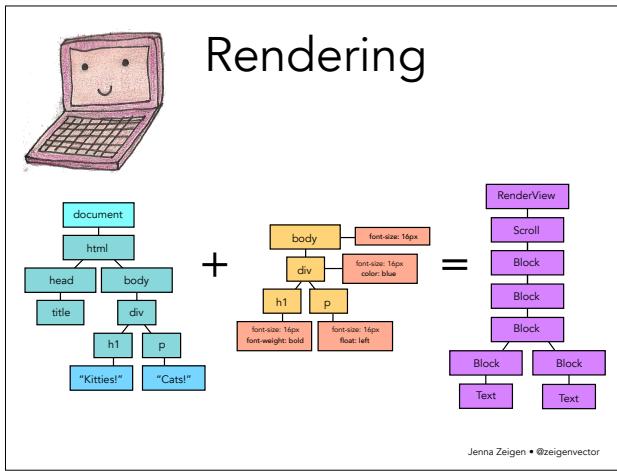
Eventually, the whole document will be fed into the parser and made into DOM tree.

## Rendering

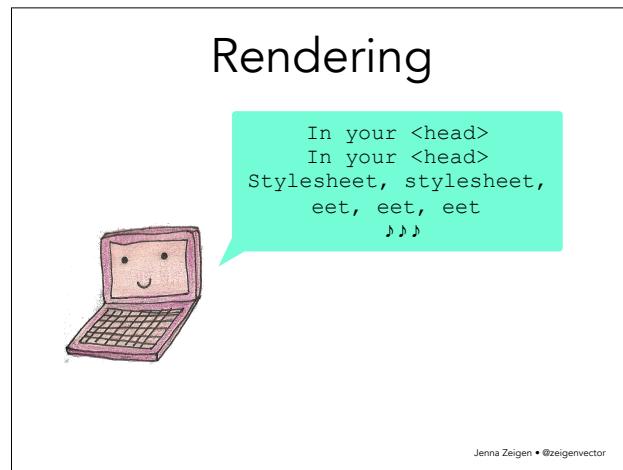


Jenna Zeigen • @zeigenvector

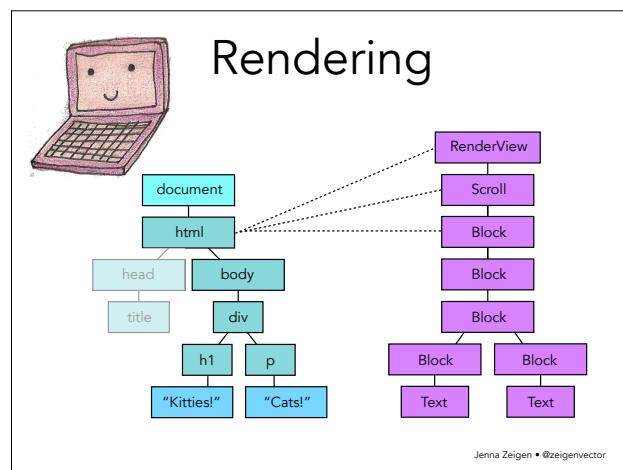
While it's important to download all the assets in the right order, rendering is just as integral a job for the browser. To kick off rendering the page, CSSOM and DOM trees are combined into a render tree, which is then used to compute the layout of each visible element and serves as an input to the paint process which renders the pixels to screen.



Because of this, HTML and CSS are both render-blocking resources, which means that the browser will hold rendering of any processed content until the DOM and CSSOM trees are constructed.



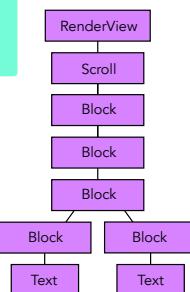
This is one reason why stylesheets are supposed to go at the top of your HTML document, in `<head>`. Putting stylesheets near the bottom of the document prohibits progressive rendering in many browsers. These browsers block rendering to avoid having to redraw elements of the page if their styles change. From a UX perspective, this to avoid the flash of unsettled content. If that's not enough of a reason, the HTML specification clearly states that stylesheets are to be included in `<head>`, so...



Elements in the render tree correspond to elements in the DOM tree, but it's not 1:1. The render tree only includes things to be rendered, so no non-visual DOM elements like `<script>` or elements with the style `display: none`. Elements with the style `visibility: hidden` will be in the render tree, because it leaves the space for it.

## Rendering

Cause the render's gonna...  
rend, rend, rend?  
♪♪♪



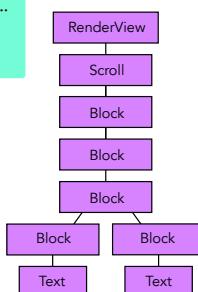
Jenna Zeigen • @zeigenvector

While the render tree contains information about what nodes are visible and computed styles, we haven't yet figured out where all these elements are supposed to go or how big they're supposed to be. This is all part of the "layout" process, also called "reflow."

Layout is a recursive process, starting at the root renderer, which corresponds to the `<html>` element of the HTML document. The browser traverses the render tree and calculates each node's position and size within the device's viewport. In this phase, all relative measurements are converted to exact pixel locations on the screen. All renderers have a "layout" or "reflow" method which lays out itself and invokes the layout method of its children that need layout.

## Rendering

Cause the render's gonna...  
rend, rend, rend?  
♪♪♪



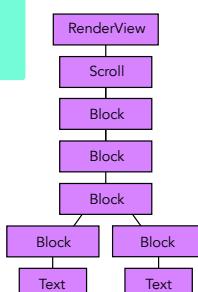
Jenna Zeigen • @zeigenvector

In order not to do a full layout for every small change, browsers use a "dirty bit" system. A renderer that is changed or added marks itself and its children as "dirty", which just means that it needs laying out. There are actually two flags: "dirty" and "children are dirty." The latter means that while the actual renderer may be "clean," it has at least one child that needs a layout.

Layouts can be either global or incremental. Incremental layouts are done in async batches based on a timer in the browser.

## Rendering

And the painter's gonna...  
paint, paint, paint...  
♪♪♪



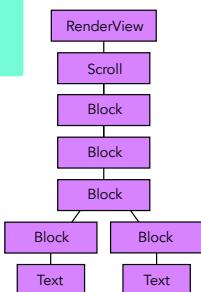
Jenna Zeigen • @zeigenvector

Once we know the location and dimensions of all the elements in the render tree, we can then start representing everything as pixels on the page — painting! In the painting stage, the render tree is traversed recursively and the renderer's "paint" method is called to display the content. This process involves the browser making layers from the bottom up in twelve phases, determining what color to paint each pixel. the paint process creates a bitmap for each layer

Like layout, painting can also be global or incremental, where only dirty regions are repainted. This is where "region-specific repainting" comes from!

## Rendering

And the painter's gonna  
paint, paint, paint...  
♪♪♪



Jenna Zeigen • @zeigenvector

Then the bitmaps are handed to the GPU for compositing (or more technically: smushing all the layers together)

## HTML Parsing

```
1 <!doctype html>
2 <html>
3   <head>
4     <script type="text/javascript" src="library.js"></script>
5     <link rel="stylesheet" type="text/css" href="styles.css">
6     <script type="text/javascript" src="website.js"></script>
7   </head>
8   <body>
9     <ul>
10       <li>Let's pretend this is all of google</li>
11   </body>
12 </html>
```



Whoa, oh, oh, oh,  
Whoa, oh, oh, oh,  
The DOM's interactive  
DOM's interactive  
♪♪♪

Jenna Zeigen • @zeigenvector

When the HTML document is fully downloaded and parsed, the page is marked as "interactive" and the `DOMContentLoaded` event is fired, without waiting for all the assets to finish loading. At this point, you can click around the page—it's interactive!

## HTML Parsing

Now that that was fired,  
fired, fired,  
We can load scripts marked  
defer, 'fer, 'fer  
♪♪♪



Jenna Zeigen • @zeigenvector

At this point, scripts marked with the `defer` attribute are executed.

## HTML Parsing



In your eyes  
The light, the heat  
In your eyes  
The DOM's complete  
♪♪♪

Jenna Zeigen • @zeigenvector

## HTTP/2



How you gonna upgrade me?  
What's higher than one point one?  
♪♪♪

Jenna Zeigen • @zeigenvector

Finally, once every resource has been downloaded and parsed, the document is marked as complete, and the `load` event is fired.

## Overview

1. IP Address Lookup
2. Opening a socket
3. Security Stuff
4. HTTP Request
5. Server Things
6. HTTP Response
7. Parsing
8. Rendering

Jenna Zeigen • @zeigenvector

HTTP/2 is a new HTTP specification that grew out of Google's work on SPDY ("speedy"). It serves to fix a bunch of the aspects of HTTP/1.1 that we need to optimize around all the time. So, once HTTP/2 is implemented across the board, most of the optimizations I discussed today will be obsolete. Features include:

- HTTP header compression (no repeated headers being passed back and forth)
- Server push technologies (sends assets with the HTML, because it knows you're gonna need it)
- Loading page elements in parallel over a single TCP connection

# Resources

## General

- [What Happens When \(Github\)](#)

## Transport

- [DNS \(Wikipedia\)](#)
- [UDP \(Wikipedia\)](#)
- [TCP \(Wikipedia\)](#)
- [TLS \(Wikipedia\)](#)



## Parsing, Rendering, and Painting

- [How Browsers Work \(HTML5 Rocks\)](#)
- [Marja Hötä: Parsing JavaScript - better lazy than eager? \(JSConf EU 2017\)](#)
- [Franziska Hinkelmann: JavaScript engines - how do they even? \(JSConf EU 2017\)](#)
- [Critical Rendering Path \(Google Developers\)](#)
- [True Grit: Debugging CSS & Render Performance \(Google I/O 2013\)](#)
- [Accelerated Rendering in Chrome: The Layer Model \(HTML5 Rocks\)](#)

Jenna Zeigen • @zeigenvector

# Resources

## Performance

- [Best Practices for Speeding Up Your Web Site \(Yahoo Developer Network\)](#)
- [Raul Fraile: How GZIP compression works \(JSConfEU 2014\)](#)

## HTTP/2

- [Andy Davies: The Case for HTTP/2 \(TXJS 2015\)](#)



Jenna Zeigen • @zeigenvector

## Complete Tracklist

The Beatles - You Know My Name (Look Up The Number)  
Carly Rae Jepsen - Call Me Maybe  
Radiohead - Creep

Taylor Swift - Blank Space  
Taylor Swift - We Are Never Ever Getting Back Together

Taylor Swift - Wildest Dreams  
Sarah McLachlan - I Will Remember You

Capital Cities - Safe and Sound  
Adele - Hello

Biz Markie - Just A Friend  
Imagine Dragons - Demons

Young Chris - Racks

Beastie Boys - So What'cha Want

Gotye - Somebody That I Used To Know  
The Beatles - All My Lovin'

Rihanna - Work

The Temptations - My Girl

Missy Elliot - Work It

Usher - Confessions, Pt. II

Calvin Harris ft. Ellie Goulding - Outside  
Jimmy Eat World - Sweetness

The Notorious B.I.G. - Sky's The Limit



Jenna Zeigen • @zeigenvector

Complete Tracklist (continued)

MC Hammer - U Can't Touch This  
 Shakira - Whenever, Wherever  
 Meghan Trainor - All About That Bass  
 The Police - Message in a Bottle  
 Rilo Kiley - The Execution of All Things  
 Mary J Blige - Not Loving You  
 Pitbull ft. Ke\$ha - Timber  
 The Beatles - Come Together  
 Jack Johnson - Better Together  
 Taking Back Sunday - Cute Without the E  
 Daft Punk - Harder, Better, Faster, Stronger  
 Taylor Swift - Blank Space (yes, again...)  
 Blink 182 - All The Small Things  
 Beyoncé - Love on Top  
 Drake - Started from the Bottom  
 Miley Cyrus - We Can't Stop  
 Spice Girls - Stop  
 Snoop Dogg & Wiz Khalifa ft. Bruno Mars - Young, Wild, & Free  
 Taylor Swift - Style  
 Britney Spears - Till the World Ends  
 Flo Rida - Low  
 R. Kelly - Ignition

\*something  
witty about a  
POP server...\*



Jenna Zeigen • @zeigenvector

Complete Tracklist (continued)

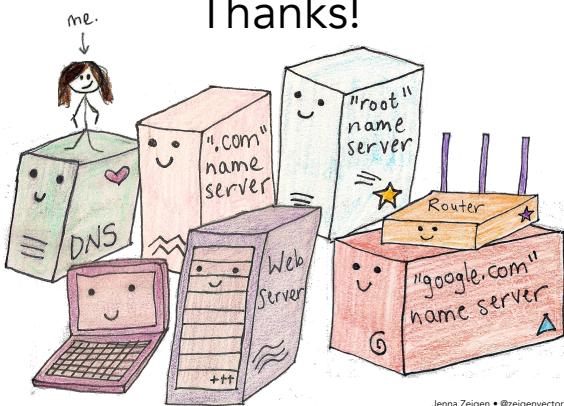
Enya - Only Time  
 Semisonic - Closing Time  
 The Cranberries - Zombie  
 Taylor Swift - Shake It Off  
 Imagine Dragons - Radioactive  
 Ellie Goulding - Burn  
 Peter Gabriel - In Your Eyes  
 Beyoncé - Upgrade U

♪ ♪ ♪



Jenna Zeigen • @zeigenvector

Thanks!



Jenna Zeigen • @zeigenvector