# Server Farm to Table

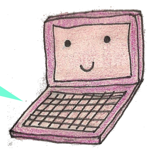
(or, how the Internet works)

Jenna Zeigen
@zeigenvector

So, what *does* happen when you type www.google.com into the browser?

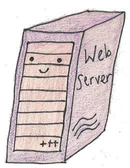
---

hey, sup?



Server Farm to Table

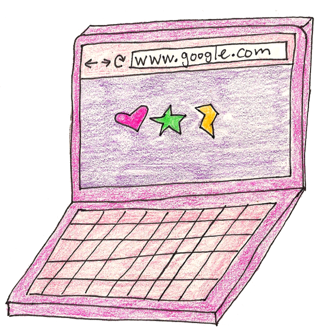
(or, how the Internet works)




nm, u?

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

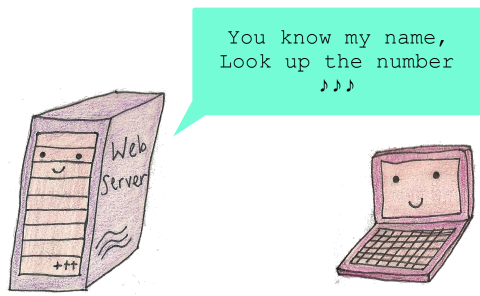
1. IP Address Lookup
2. Opening a socket
3. Security Stuff
4. HTTP Request
5. Server Things
6. HTTP Response
7. Parsing
8. Rendering

Jenna Zeigen • @zeigenvector

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

---

# IP Address Lookup

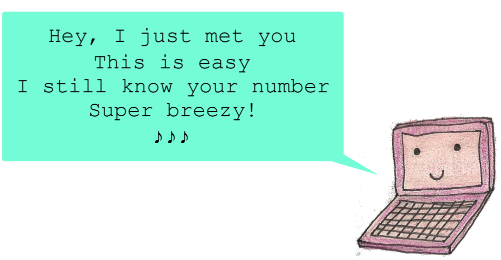



Jenna Zeigen • @zeigenvector

To do anything, we need to map the human readable URL "www.google.com" to the address of a computer to "talk to."

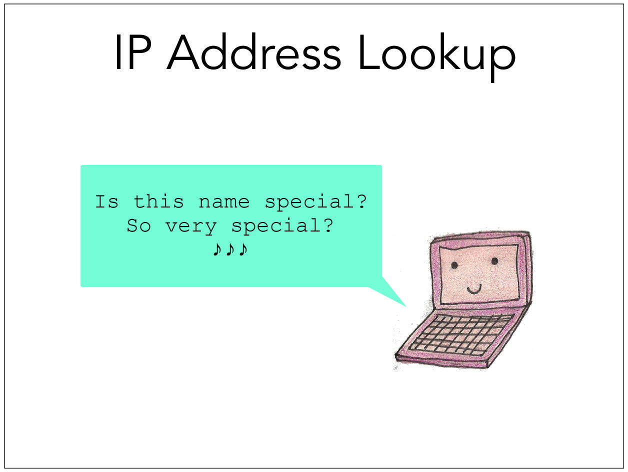
---

# IP Address Lookup

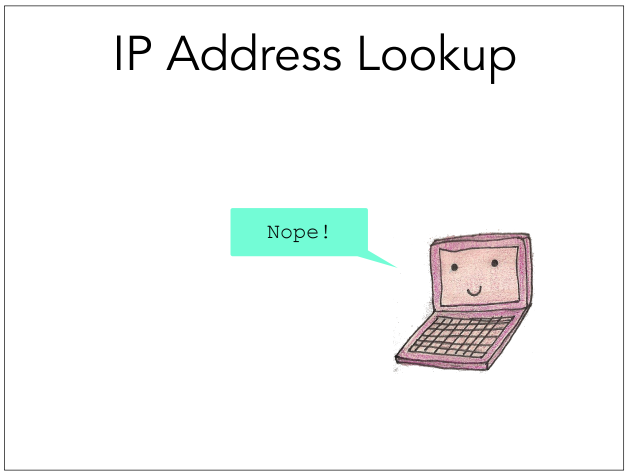



Jenna Zeigen • @zeigenvector

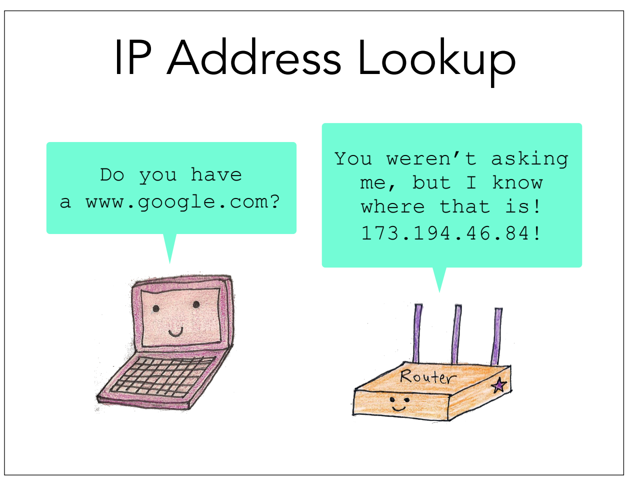
First, the browser checks its cache.



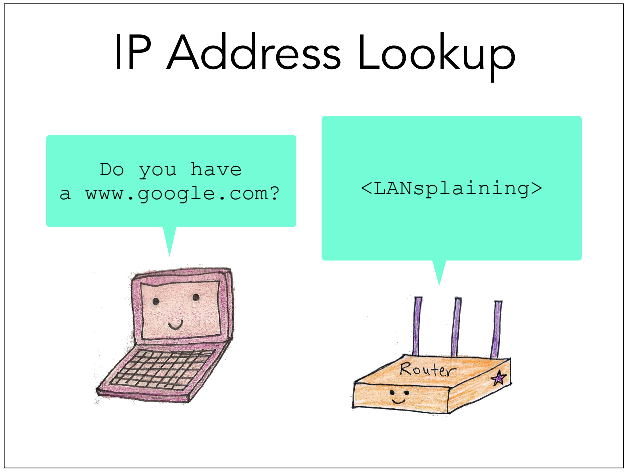

If not found, the browser calls the `gethostbyname` library function, the contents of which vary by OS, to do the lookup. This function checks if the hostname is listed in the local hosts file, the location of which also varies by OS. On Macs, it's at `/etc/hosts`. This is how your computer knows what 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!



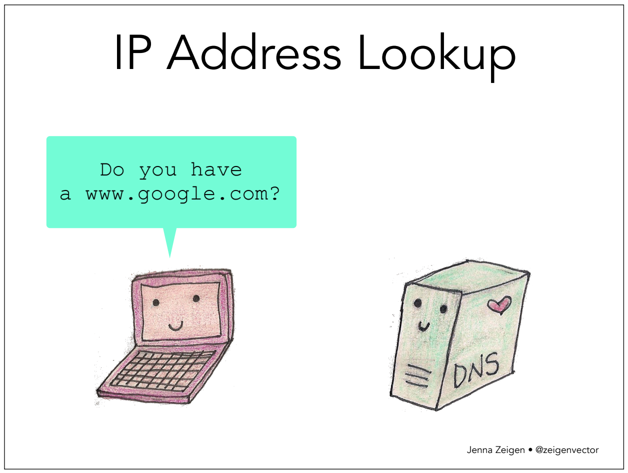

If `gethostbyname` does not have it cached and can't find it in the hosts file, then it makes a request to the DNS server configured in the network stack. This is typically the ISP's caching DNS server, but you could configure your computer to use any DNS server, i.e. Google's (8.8.8.8) if you wanted…




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.

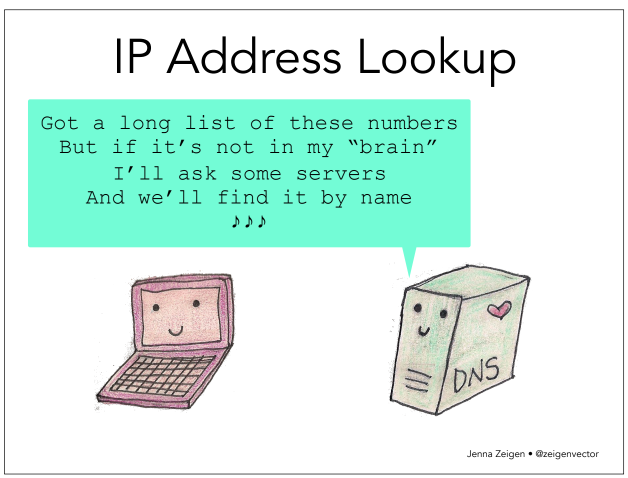



(Big thanks to @jedschmidt for this witty commentary)

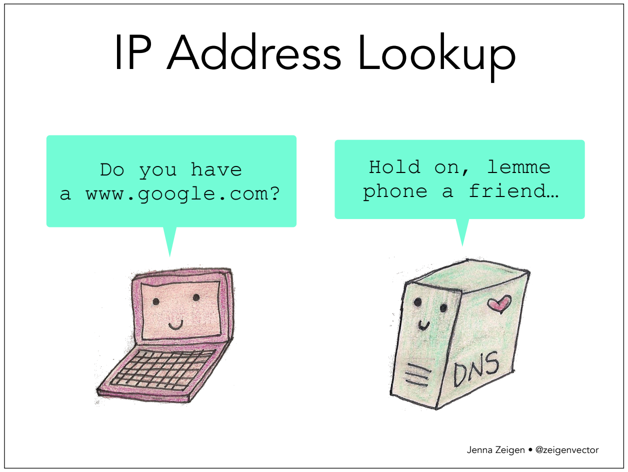



If the local router doesn't have anything cached, the browser ends up hitting the local ISP DNS server. This process uses UDP (User Datagram Protocol), or TCP if the request is too large. Communications protocols are ways that entities transmit information among them, and we'll hear more about TCP later.

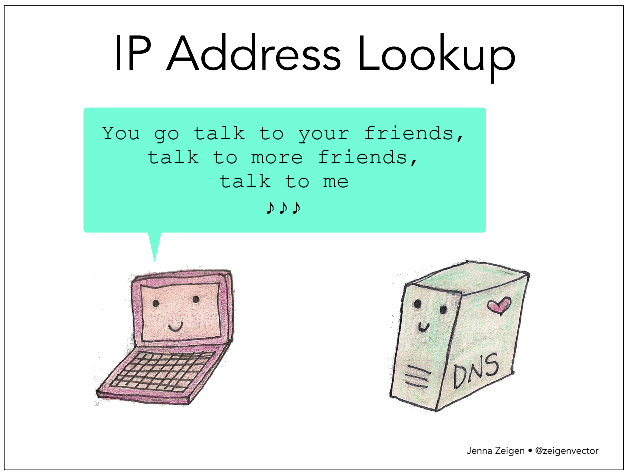
UDP is kind of like the smallest bit of network you can use. It's great for unidirectional transactions, but delivery is not guaranteed in any order, or guaranteed at all. It's kind of like if you were to through a bunch of balls at someone and hope that they catch them.



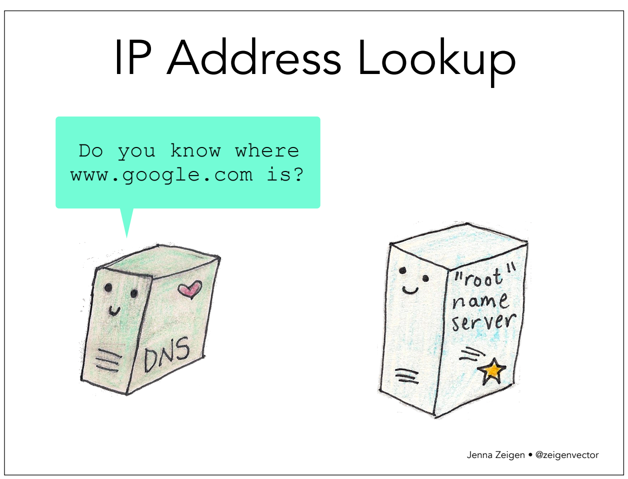

Upon receiving the request, the DNS server first checks its cache.

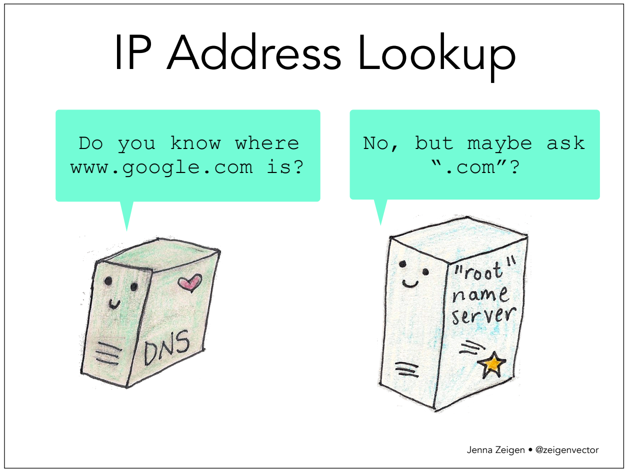


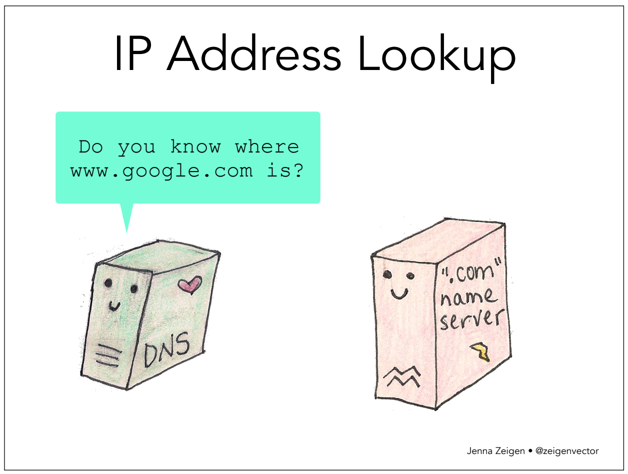

If it doesn't have the address in its cache, it recursively looks up the address by asking some more servers…

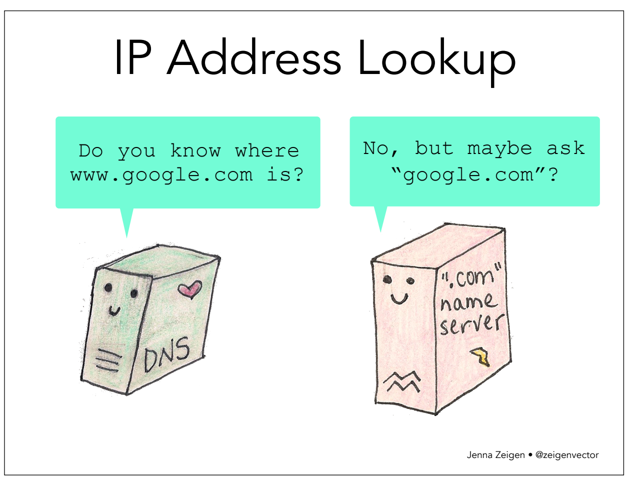


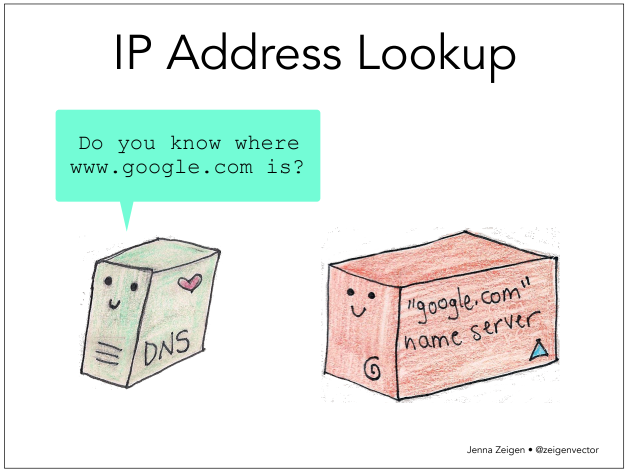

(Meanwhile, the client has to wait a while for the answer…)

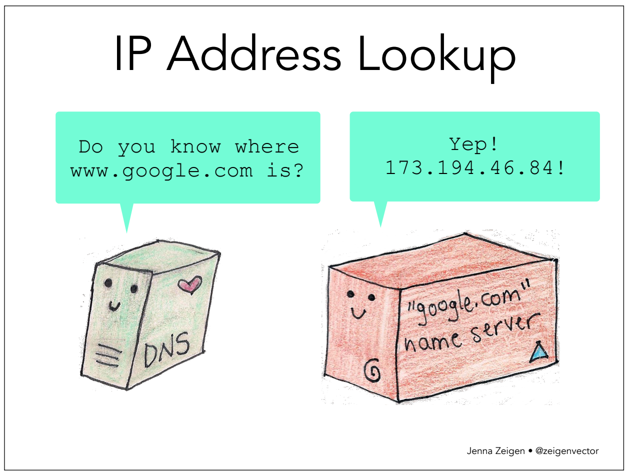


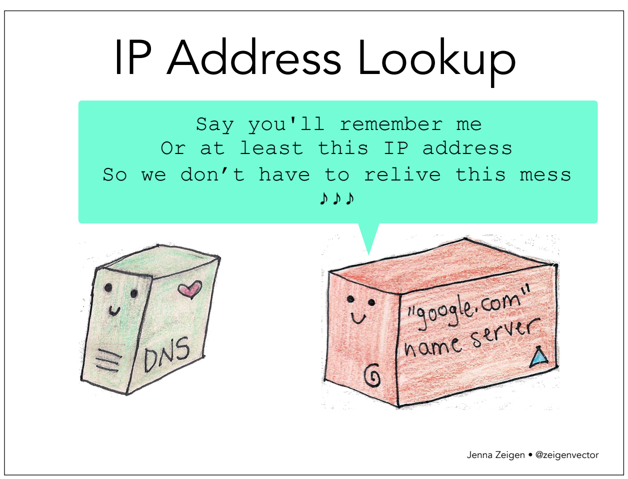

IP Address Lookup
Do you know where www.google.com is?
No, but maybe ask ".com"?
DNS
"root" name server
Jenna Zeigen • @zeigenvector


IP Address Lookup
Do you know where www.google.com is?
DNS
".com" name server
Jenna Zeigen • @zeigenvector

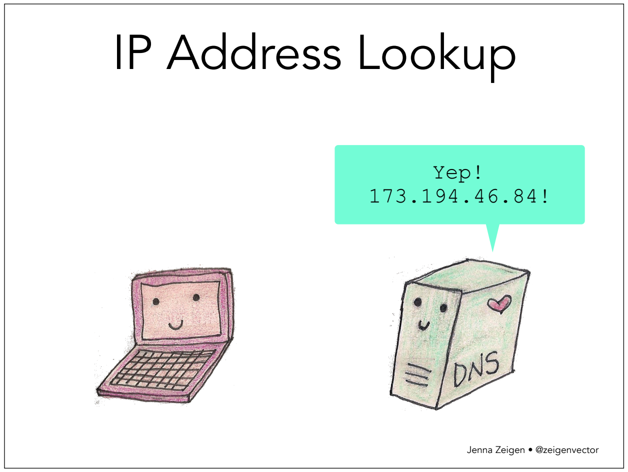

IP Address Lookup
Do you know where www.google.com is?
No, but maybe ask "google.com"?
DNS
".com" name server
Jenna Zeigen • @zeigenvector



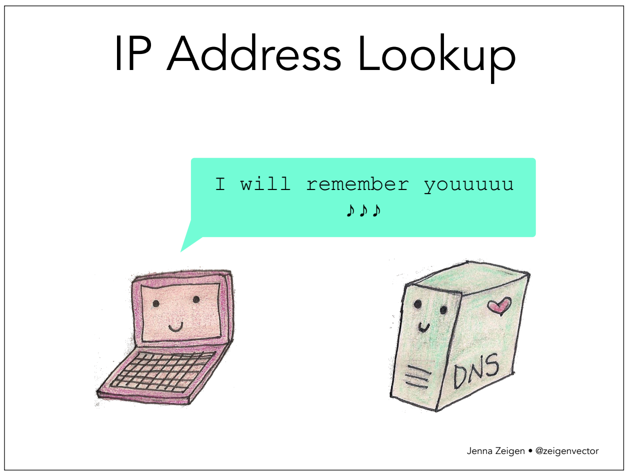



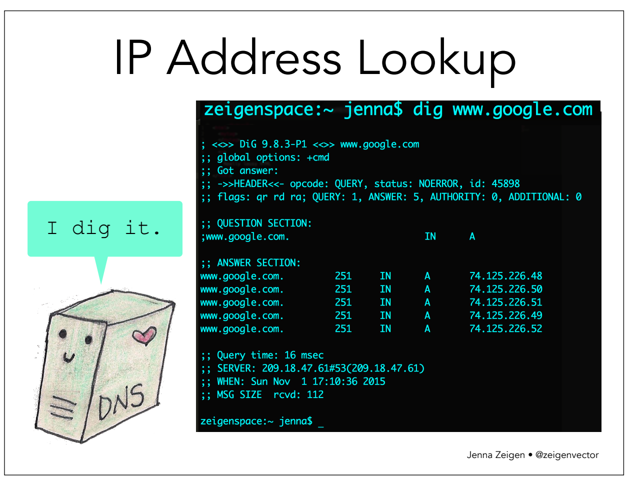

When the name server comes back to the DNS server with the IP address answer, it will also come back with a TTL (time to live), an amount of time that the DNS server should cache the IP address. The longer the TTL, the faster this process, but then it's harder to change things. Historically, this TTL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



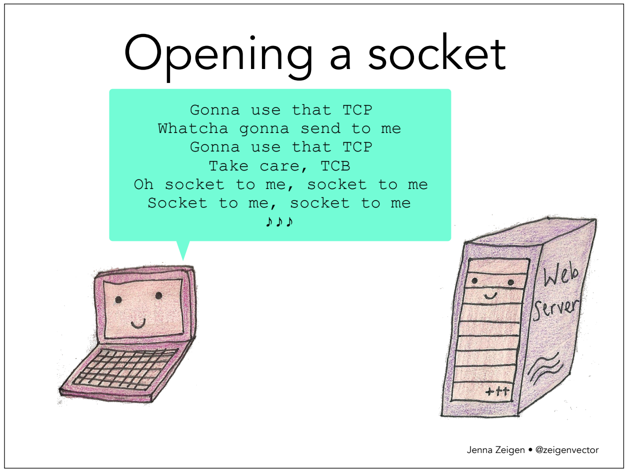

The DNS server then responds to the client with the found IP address.



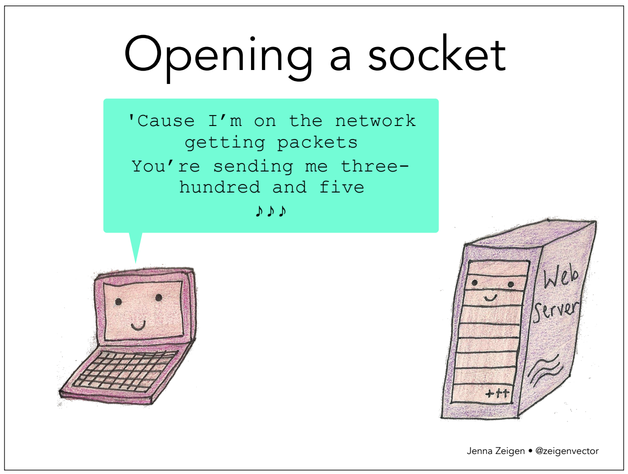

The client then caches the IP address for next time. Some browsers also have their own DNS cache.



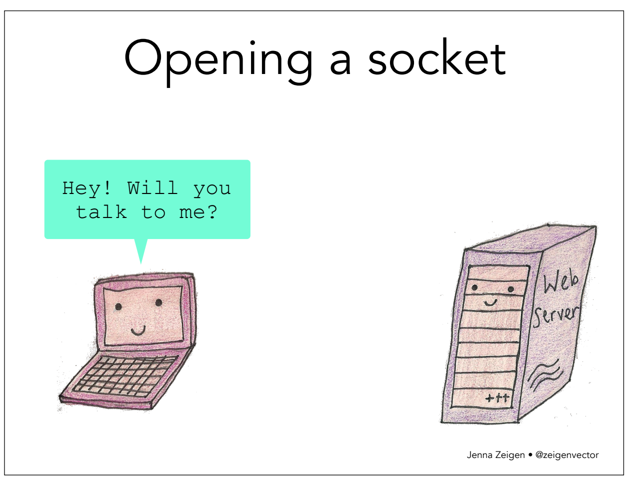

If you want to check out this process hands-on, you can use the `dig` command in your terminal.



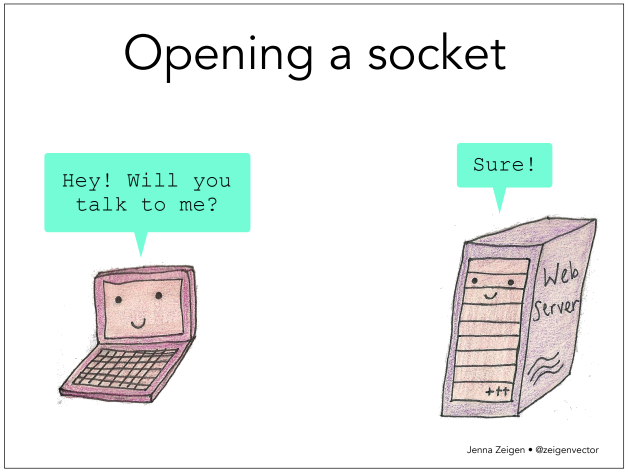

Once the client has the IP address, it takes it and the port number from the URL, and makes a call to the system library function named `socket` and requests a TCP socket stream. We don't usually see port numbers on the end of URLs, but the HTTP protocol defaults to Port 80, and HTTPS to Port 443.

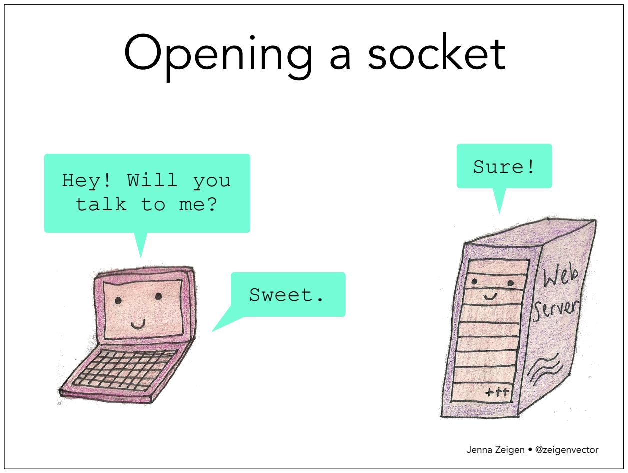


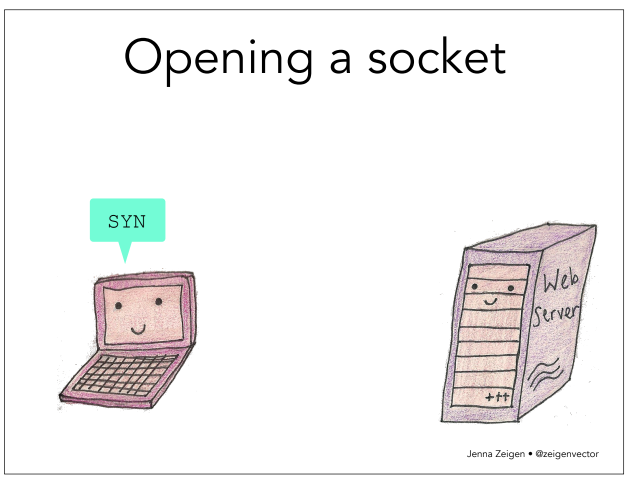

TCP stands for "Transmission Control Protocol." Unlike UDP, its job is to deliver a reliable, ordered, and error checked stream of packets across the network. It's more like a game of catch, to continue the ball metaphor. In fact, it's often called the "TCP handshake". TCP is the protocol that underlies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




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

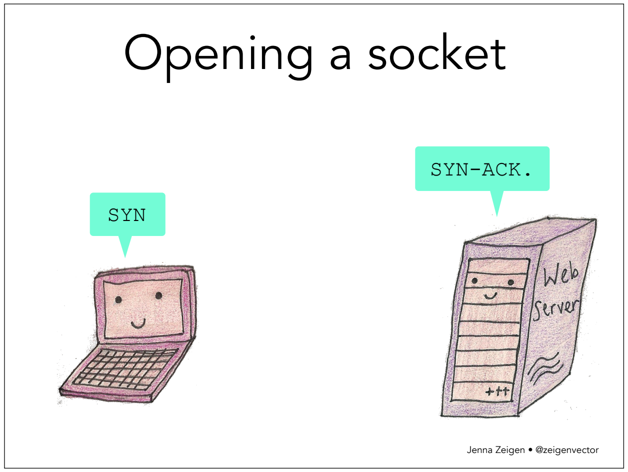


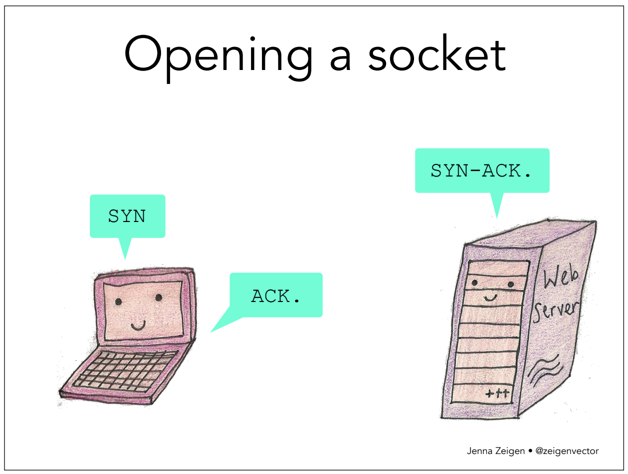




Well, more specifically like this…

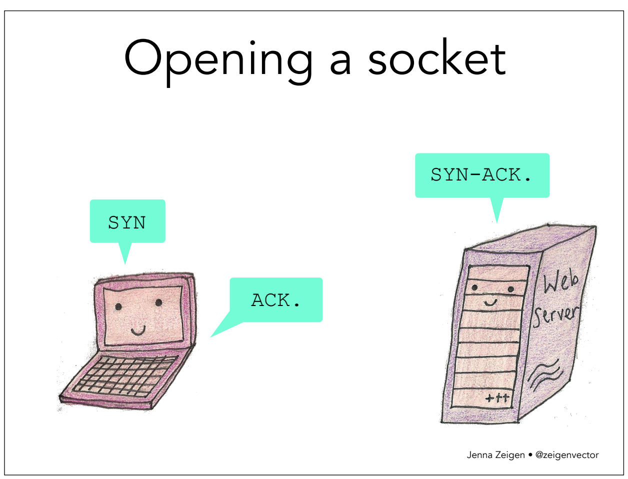
1. The client sends a SYN packet with a randomly-selected sequence number "x"



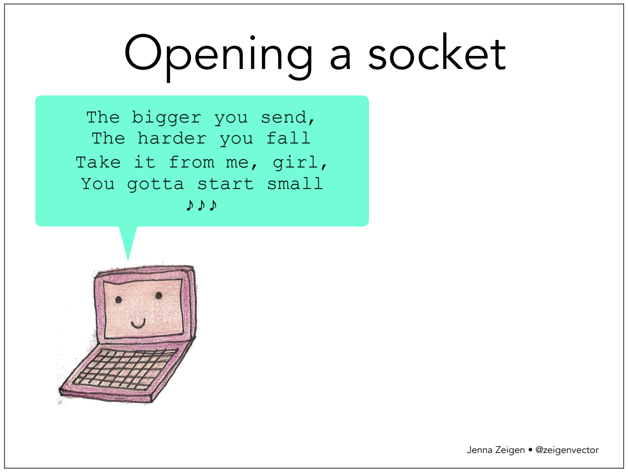

2. The server replies with a SYN-ACK packet with an acknowledgement number set to “x+1” and a different randomly-selected sequence number “y”



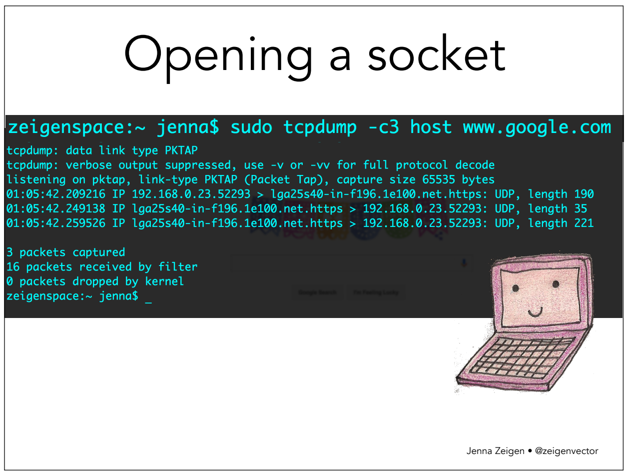

3. The client responds with an ACK packet, with the sequence number set to the received acknowledgement number “x+1”, and a new acknowledgement number set to “y+1”



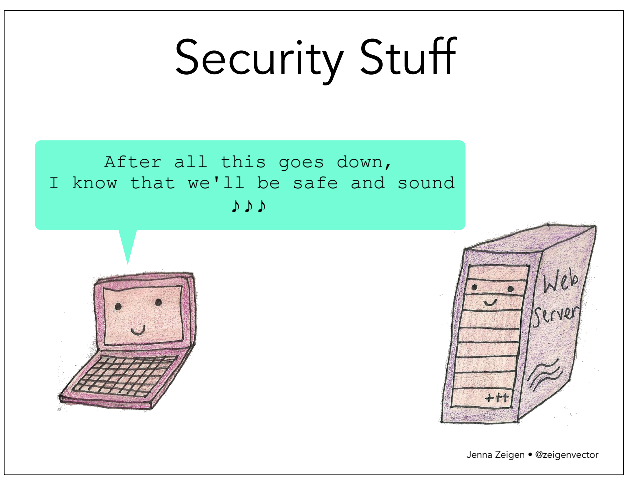

To prepare for the likely event that the client is going to need more things from the server, more than one connection will be opened.



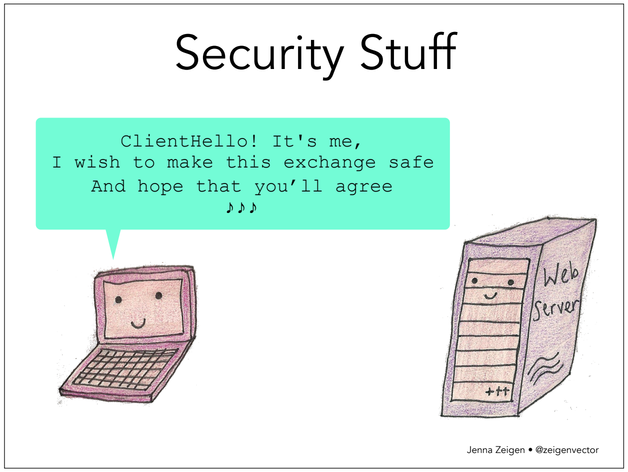

TCP also has congestion control— requests start small and ramp up in size to make sure the network can support the requests. This is more of a relic from the past when networks weren't as reliable or fast. TCP is where much of the latency in this entire process comes from, as it's constrained by the speed of light. This is why it is advantageous to serve assets from a server as close to your user as possible by using a CDN.



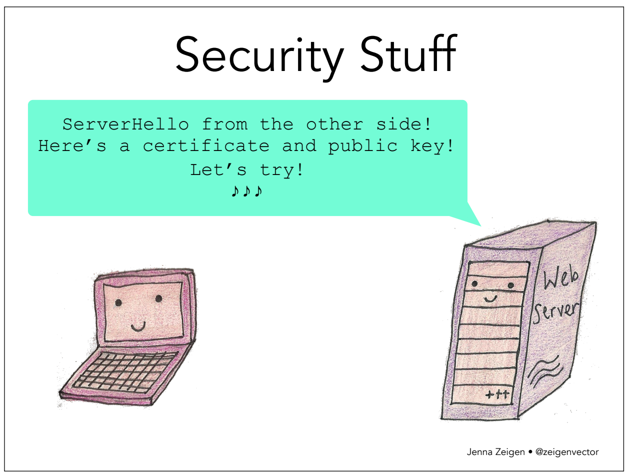

If you want to check out TCP for yourself, you can use `tcpdump` from your terminal. `tcpdump` will show you all the network traffic, which is why you see UDP packets here. If you prefer a GUI option, you can also use Wireshark.



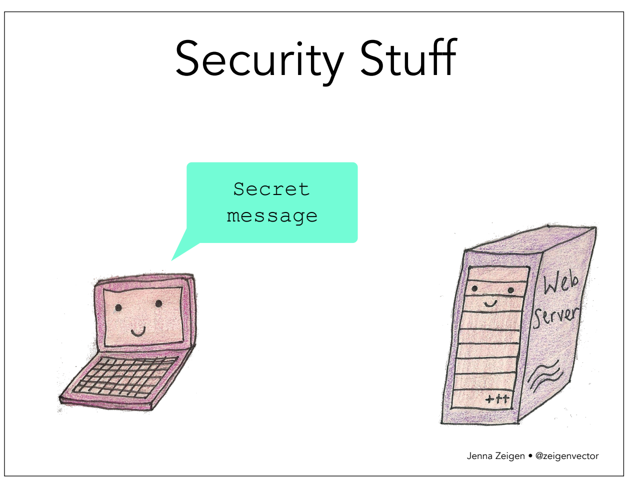

If the request is over HTTPS, we need another step to ensure the security of the transaction. HTTPS uses TLS, or Transport Layer Security. TLS's predecessor was Secure Sockets Layer (SSL).



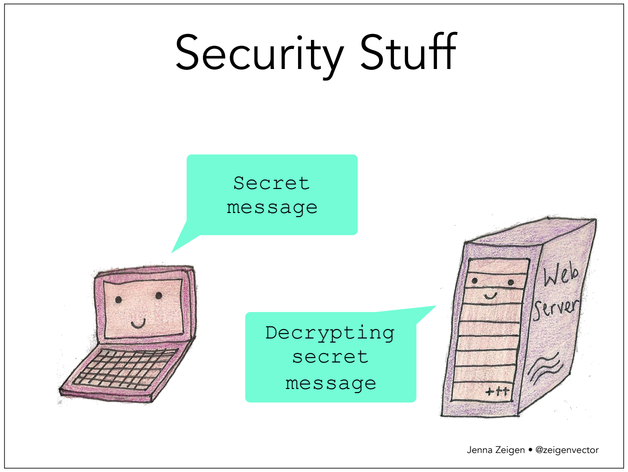

To kick of the process, the client computer sends a ClientHello message to the server with its TLS version, list of cipher algorithms and compression methods available.



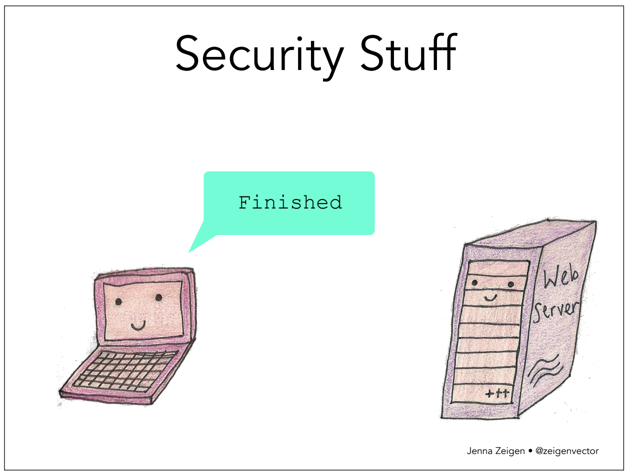

The server replies with a ServerHello message to the client with the TLS version, selected cipher, selected compression methods and the server's public certificate. The certificate contains a public key that will be used by the client to encrypt the rest of the handshake until a symmetric key can be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



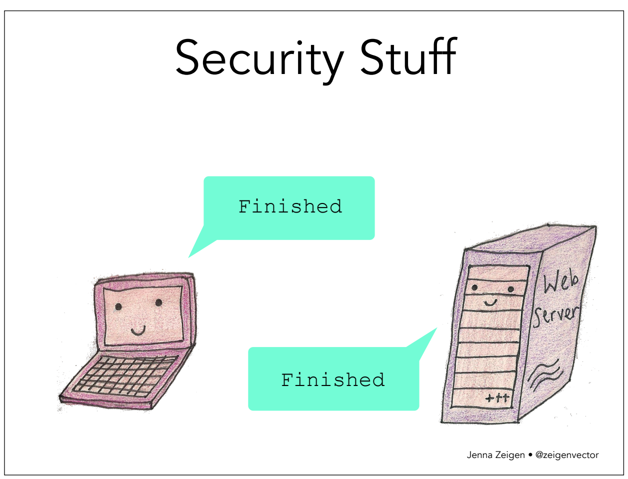

The client verifies the server digital certificate. If trust can be established, the client generates a string of pseudo-random bytes and encrypts this with the server's public key.



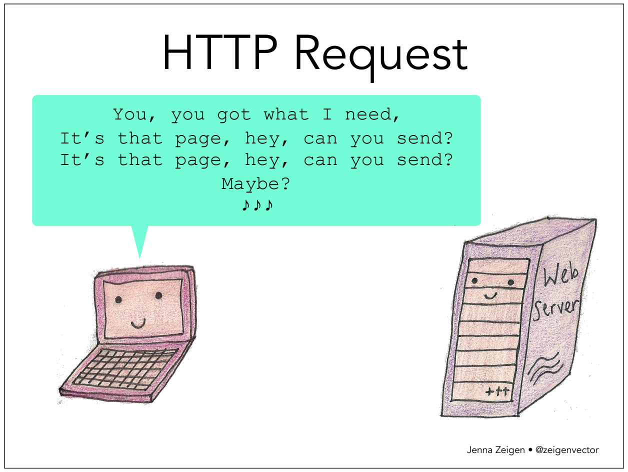

The server then decrypts the message and uses it to create its own version of the symmetric key.




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

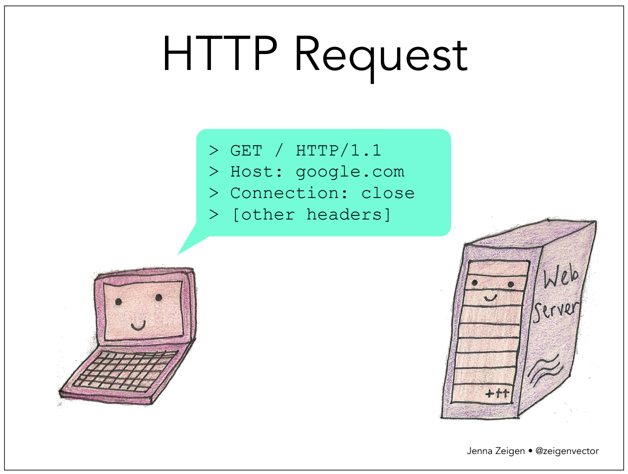



The server generates its own hash, and then decrypts the client-sent hash to verify that it matches. If it does, it sends its own Finished message to the client, also encrypted with the symmetric key.

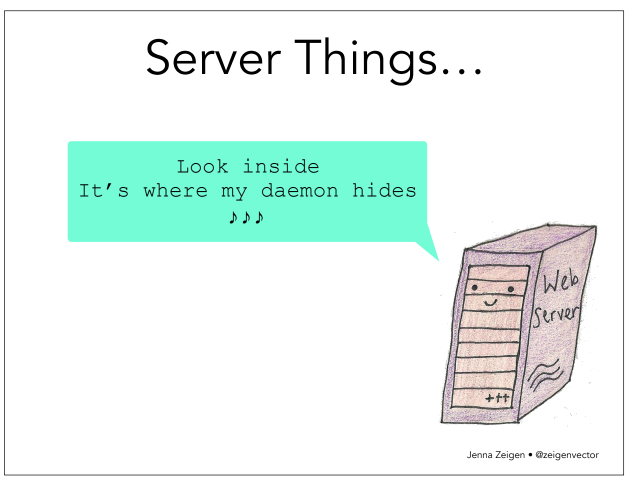



With all this done, the client is ready to send an HTTP request to the server. HTTP stands for "HyperText Transfer Protocol." Unless the client and server negotiate using a more advanced version of HTTP, they're going to use HTTP/1.1.

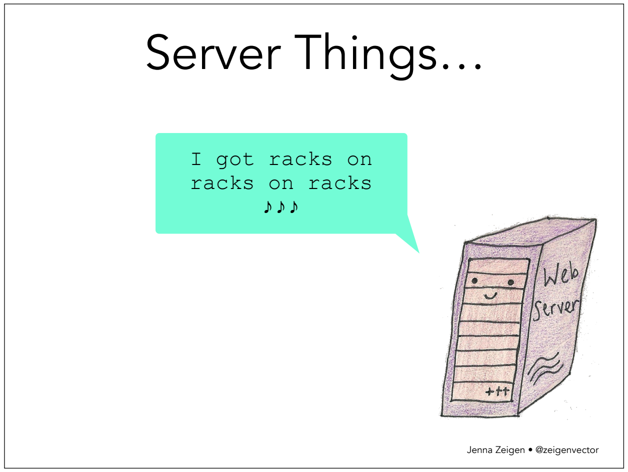
HTTP/1.1 is stateless, uses a request/response model, and is a plain-text protocol.



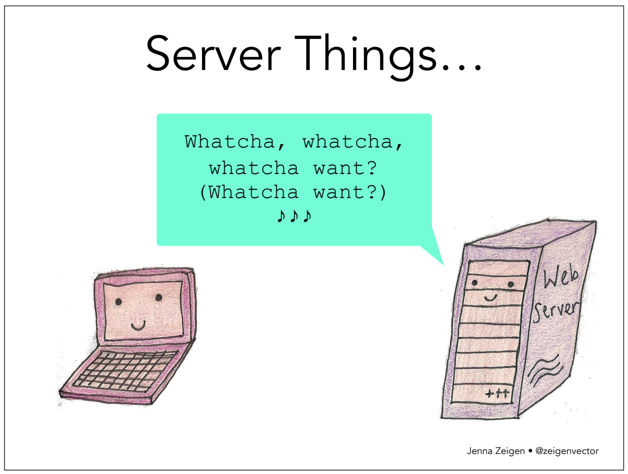

The client will send an request like this, containing the request and headers, followed by a single blank newline to the server indicating that the content of the request is done.




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.

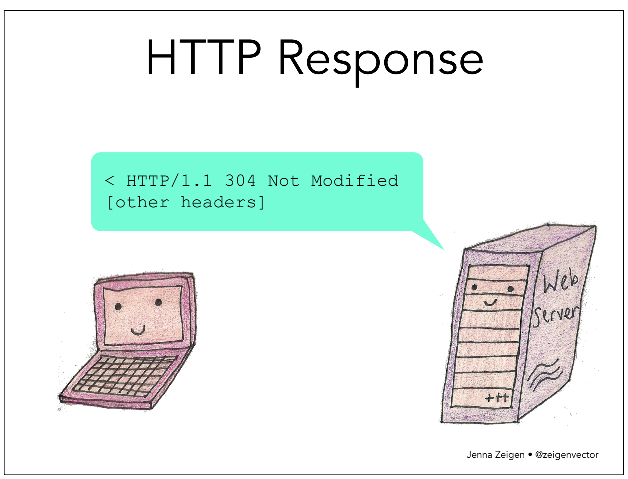



The server is going to do a bunch of "server stuff" including load balancing, routing, and database calls.



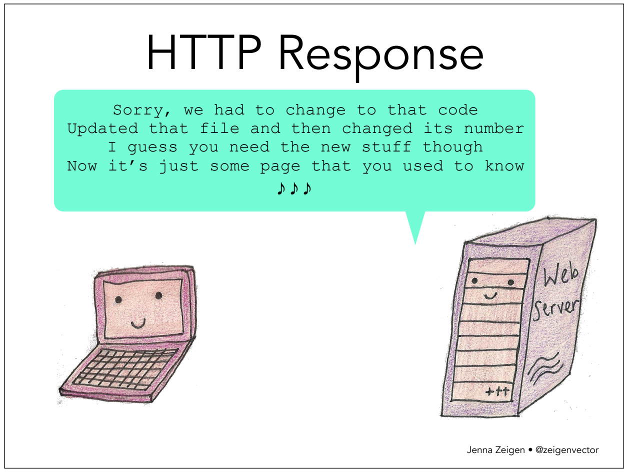

The server breaks down the request to the following parameters:

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."
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).

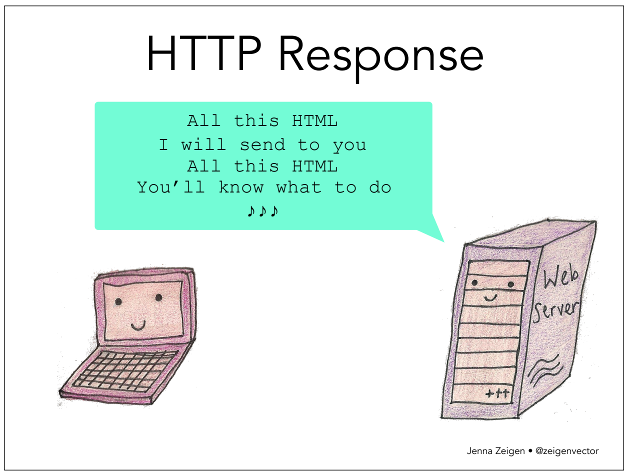



If you've been to this site before, and the HTTP headers sent by the web browser included sufficient information for the server to determine if the version of the file cached by the web browser has been unmodified since the last retrieval, it may instead respond with something like this.

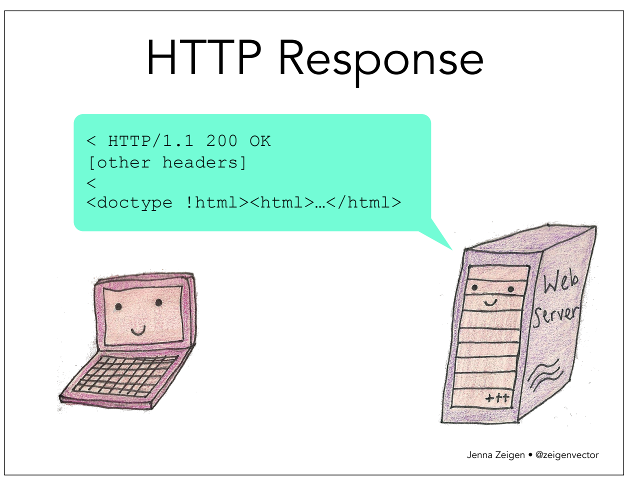
This is so much less work for the client and server at this point, but is the result of author work, making sure that the proper mechanisms are in place for the required data to be transmitted.



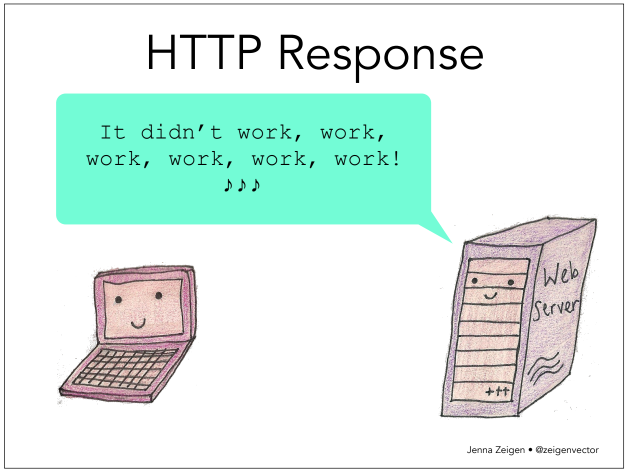

For example, having ETags in the header will allow the server to determine if the cached version of a page is still valid. The first time the server sends the page, it will send an ETag, a unique identifier for the version of the page, in the header of the response. The client will then cache this. The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



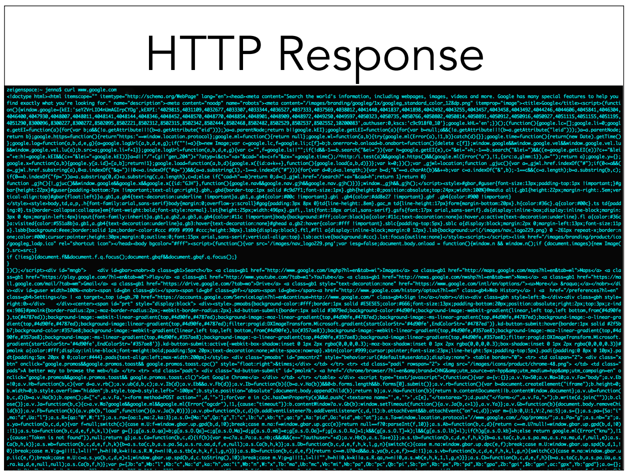

If they don't match, then there's much more to be done. The server has to then send over a much bigger payload.



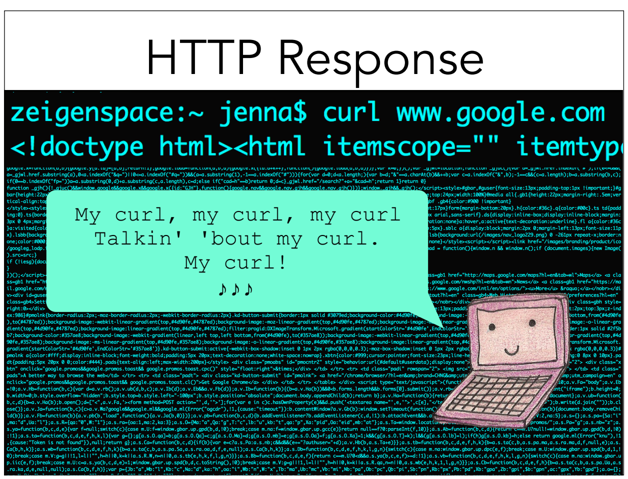

An otherwise successful request and smooth servering process will prompt the server to send a response like this, with a 200 OK status code, other headers, a single blank newline, and then the payload, which in this case is the entire HTML document.

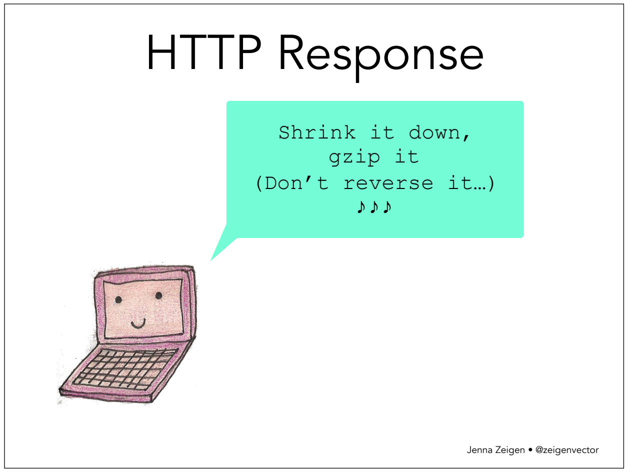



If something went wrong, you could also get less-happy status codes. The 400-series represents a client error like `403 Forbidden` or `404 Not Found`. The 500-series represents server errors, such as the `500 Internal Server Error`.

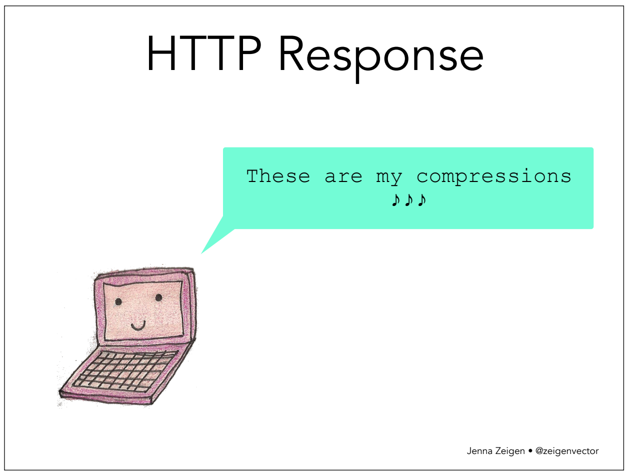



If you want to investigate what's going on in an HTTP request, you can use `curl` from your terminal.



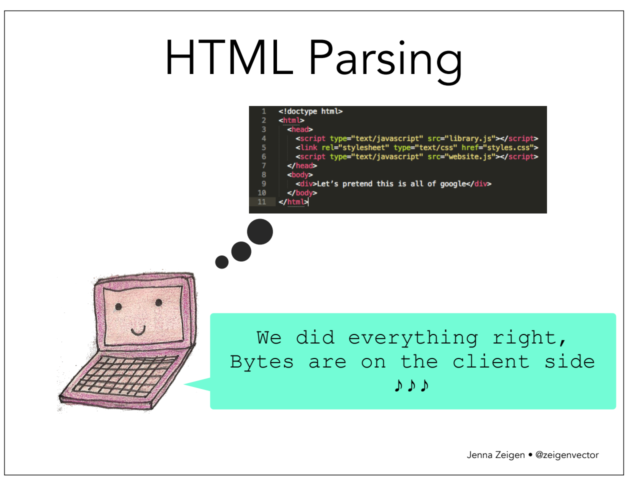



With HTTP, the size of the request directly correlates to the time it takes to finish the request. Because of this, compression helps a lot to get your page to the user as fast as possible. Gzip is most popular compression method for these web things. Gzip works by replacing repeated substrings with 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%.

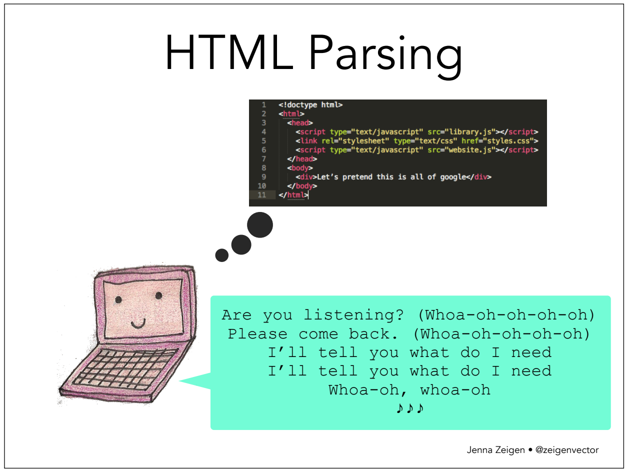



And if you're worried your users don't have this fancy compression method enabled in their browsers, the server and client talk about using gzip beforehand to make sure it's safe. And about 90% of today's Internet traffic travels through browsers that claim to support gzip anyway.

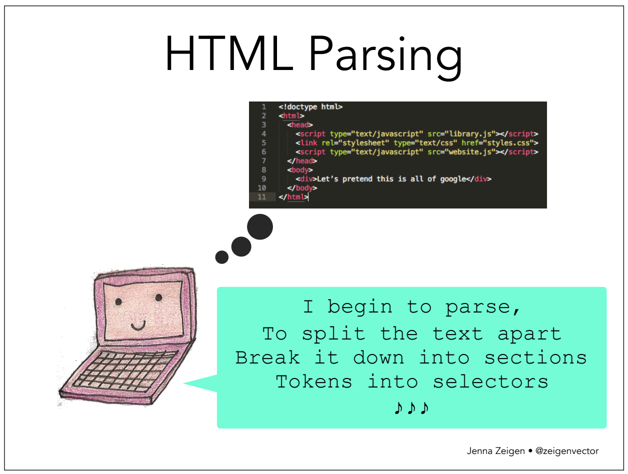
(Thanks to @jamessocol for this one)



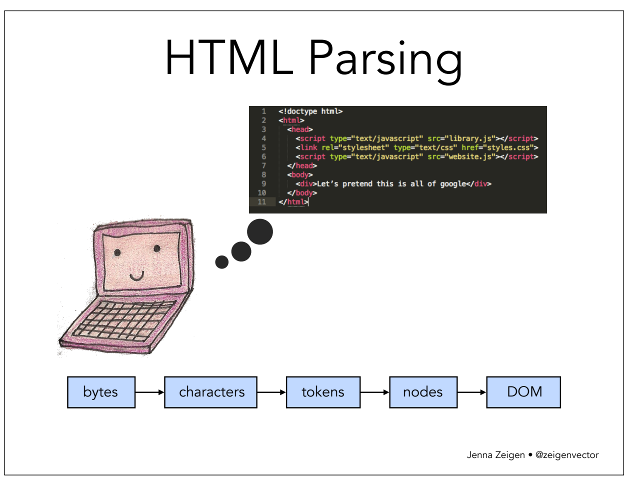

As soon as the browser starts receiving bytes, it starts parsing it. It usually receives it in 8kB chunks, and feeds those bytes right into the parser.




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%.

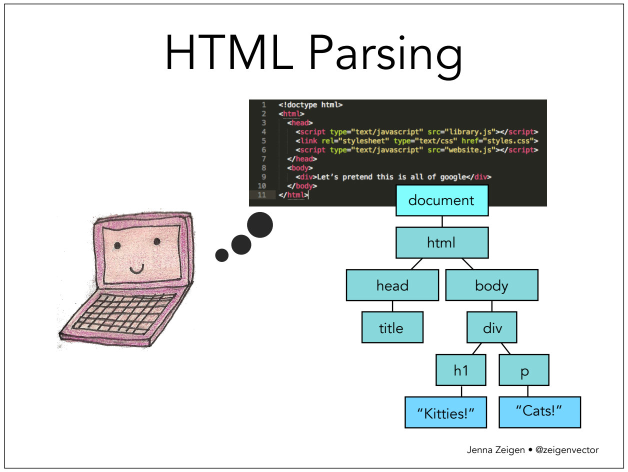



Meanwhile, the bytes are fed into the primary parser.

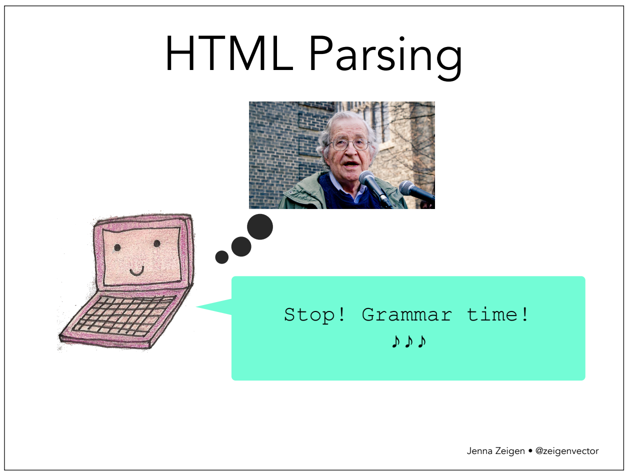



The parsing algorithm is described in detail by the HTML5 specification. It goes something like this:

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.

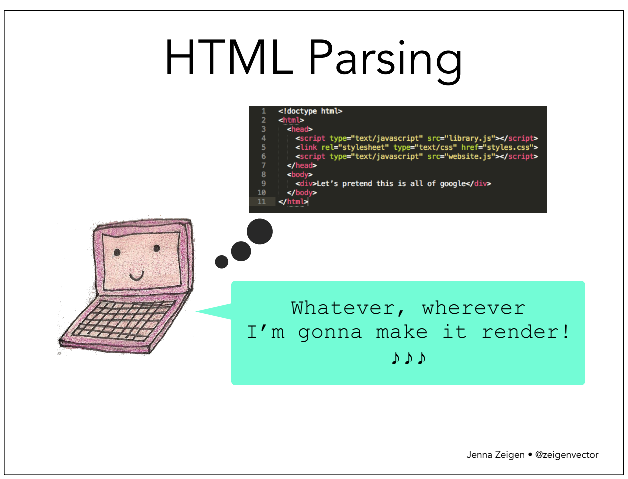



After this process, you end up with a DOM (Document Object Model) tree made from the HTML document. It has an almost one-to-one relationship with the markup.

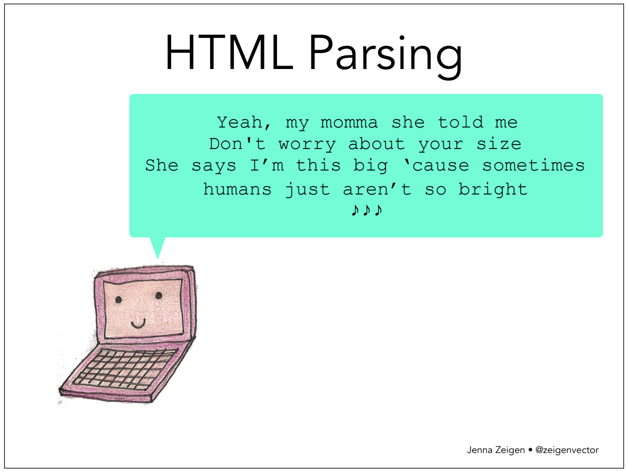



Most programming languages have a vocabulary described using regular expressions and a syntax described by a context-free grammar. A context-free grammar means there is a set of production rules that describe all possible strings in the language. Production rules are simple replacements that can be applied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

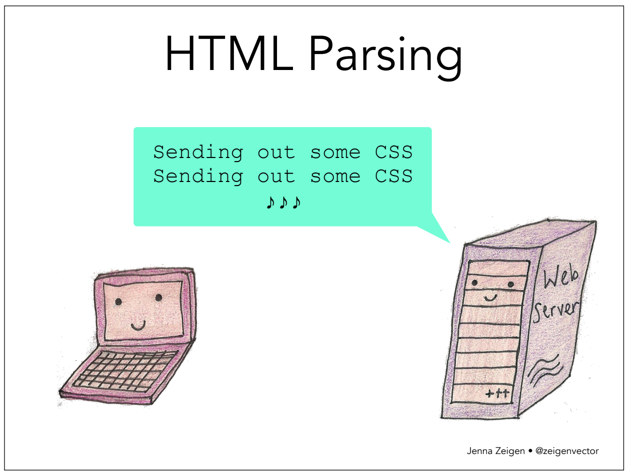
(Photo credit: https://www.flickr.com/photos/andrewrusk/5599576288)



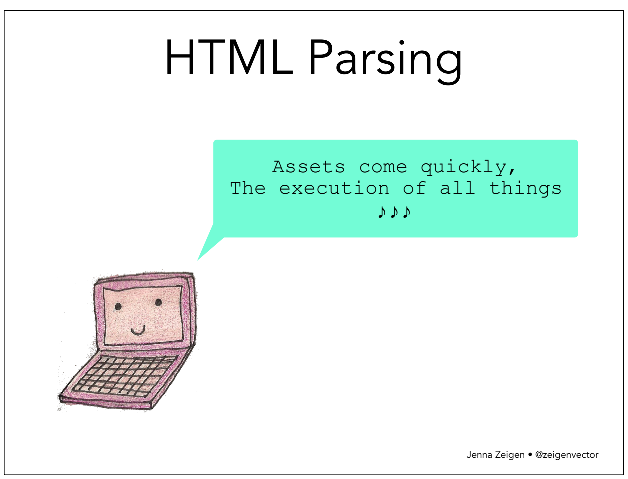

HTML isn't defined by a context-free grammar and therefore cannot be parsed by a "regular" parser. This is because HTML is designed to be more forgiving of human error— the spec requires the parser to be 100% fault-tolerant. This is why you never see an HTML syntax error. It will always render something (but sometimes it's not *exactly* what you wanted…). Thus, a lot of the parser's code is for fixing the HTML author mistakes, like invalid tags, unclosed tags, incorrect nesting.



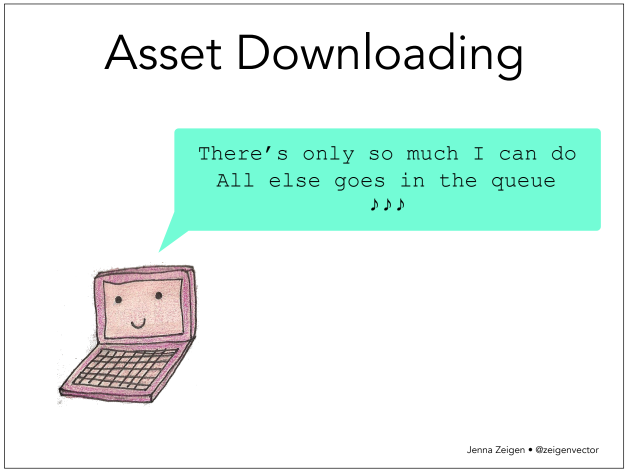

This means the parser will never error, but it also means that browsers makers have to create custom parsers. Browsers are like a million lines of C++, and a big chunk is related to this parser.



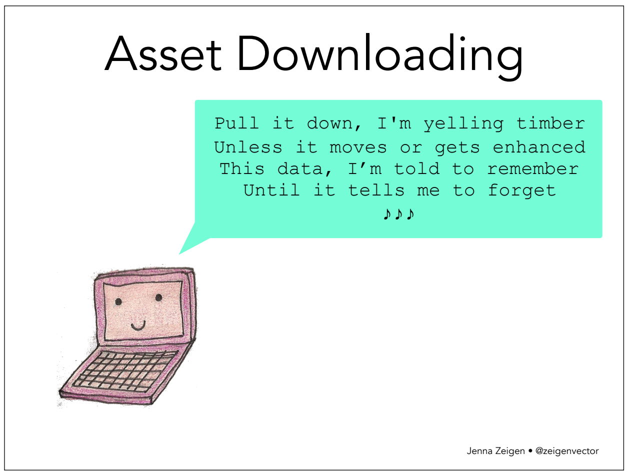

As the parser is parsing, it runs into inlined assets like images, scripts, and stylesheets. The client must then request each asset and wait for the server to send it back. Hopefully the speculative parser has already started the fetching process!



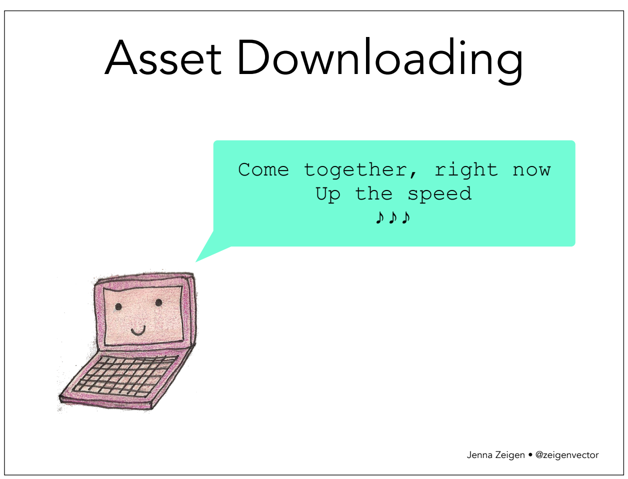

Once the browser has the asset, it then must parse and execute it before anything continues.



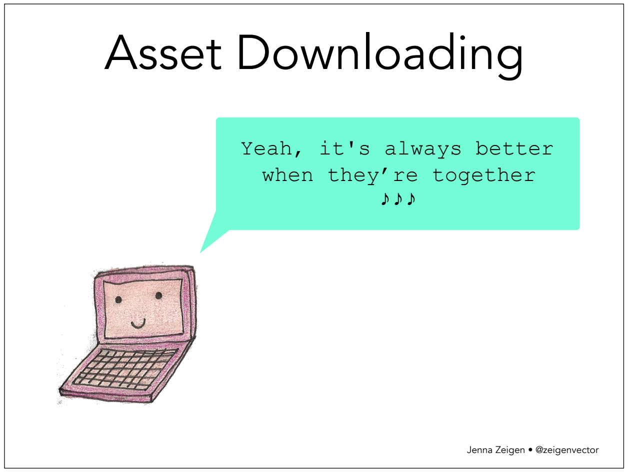

The big hiccup here is that browsers have imposed limits on how many parallel downloads you can have from the same domain. It used to be 2, but now that limit is anywhere from 6 to 13 in modern browsers, depending on the browser. This is a huge bottleneck, but many of the optimizations we hear about 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.



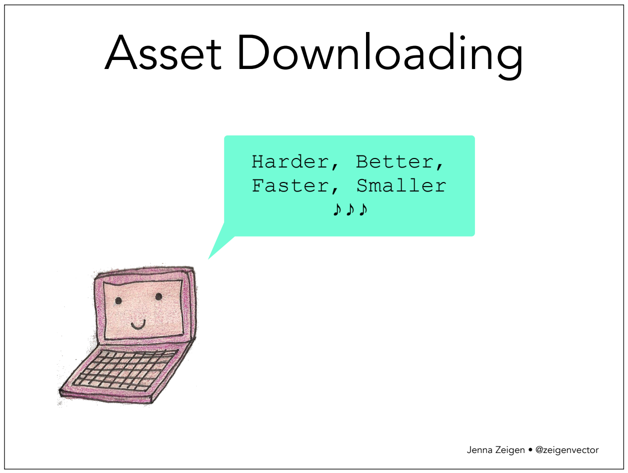

Caching also comes into play here, because TTL in this case is determined by the headers sent with the asset, such as a far-future Expires header. This helps because the file does not need to be downloaded again, therefore not contributing to the download limit.



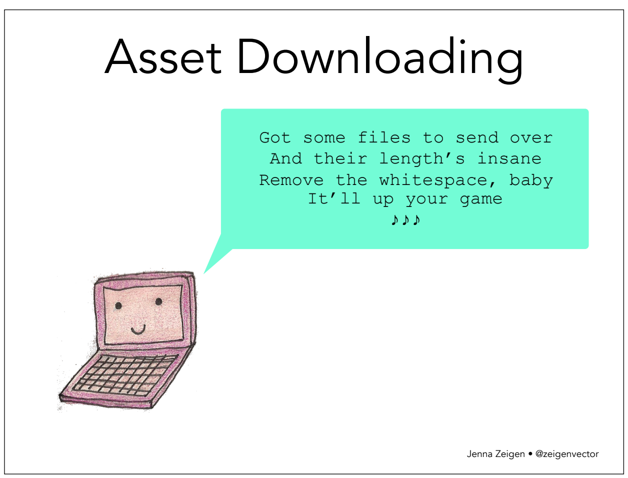

Another popular optimization is to combine your files together. This includes concatenating JavaScript and CSS files together using a tool like webpack, Gulp, or even the asset pipeline. However, this might come back to bite you with caching because overtime you change just one file in your bundle, the 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.



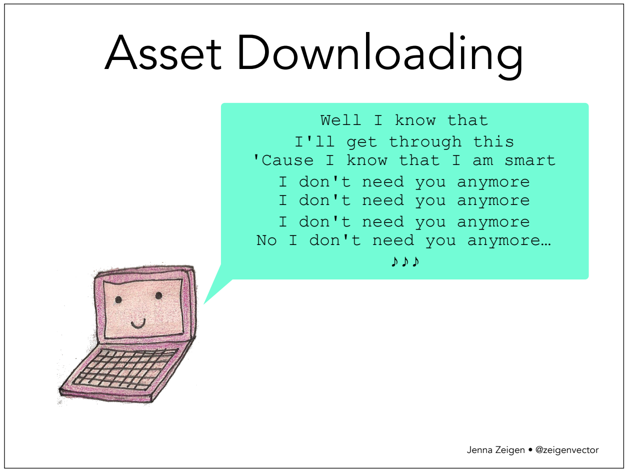

Another thing you can do is make CSS sprites or icon fonts so there aren't so many small images each with their own requests.




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.

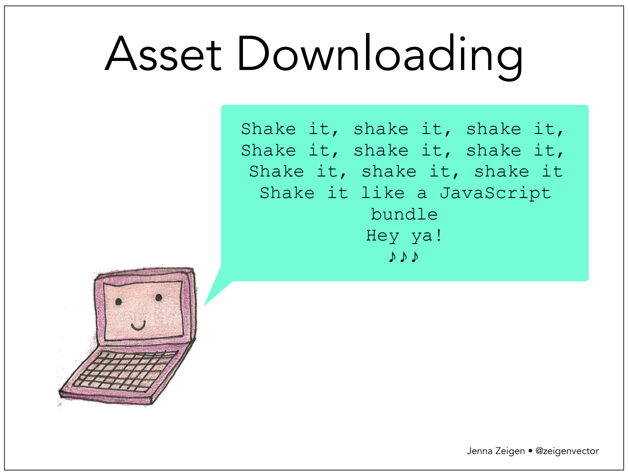



One thing you can do is minify all your HTML, CSS, and JavaScript. While it's good to write your code for humans with whitespace and comments, computers don't care about all of this. Minification gets rid of all the stuff we put in code for humans.

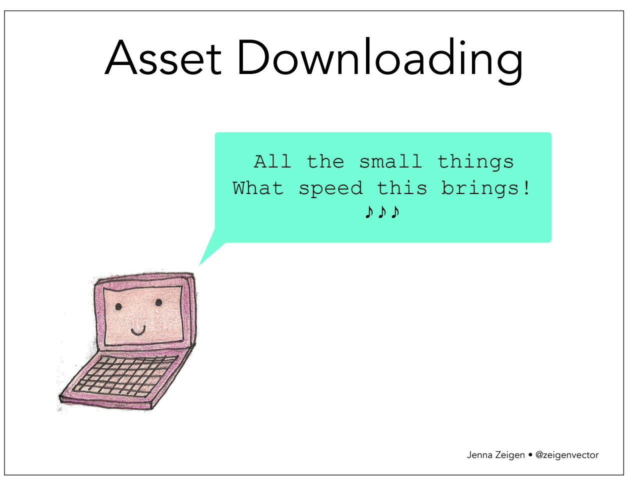



Carefully consider the impact of libraries and frameworks on both site size and developer productivity. Not every project needs a framework. If you do, choose carefully. Consider at least the total cost on size and initial parse times before choosing an option.

On a smaller scale, libraries have split what was previously a monolith into discrete modules. Right now I'm thinking of LoDash but there are likely others. Taking LoDash as an example, it's likely you're not using every single module it has to offer, so import only what you're using and not the entire library.

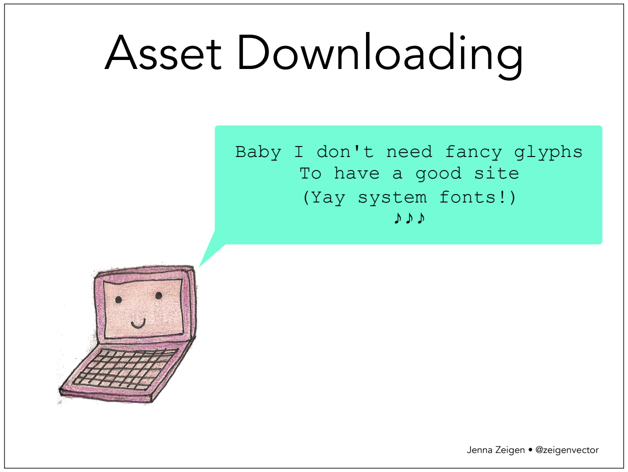



Tree-shaking is also a good way to reduce the size of the files you're sending over the wire. Tree-shaking is a method for dead-code elimination, though Rich Harris, the developer of Rollup, the module bundler that re-popularized the term in the JS community, prefers to call it "live code inclusion". To do this, module bundlers like Webpack and Rollup rely on the import and export statements in ES2015 to detect if code modules are exported and imported for use between JavaScript files. It goes through and figures out which code is going to get used and includes only that in the bundle.

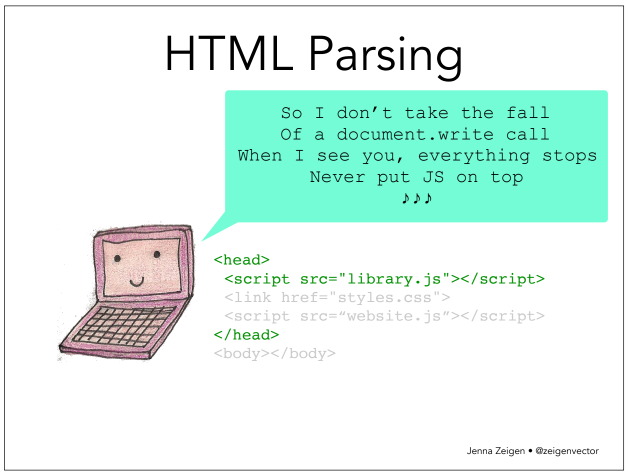



Optimizing images is also important. JPEGs can be huge, especially for mobile devices, and SVGs especially have extra information put in by the program that made the SVG such as Sketch or illustrator, as well as whitespace like we removed from our earlier files though minification. Running these images through an optimizer such as svgo or TinyPNG removes extra information like the whitespace from SVGs, or extra data from JPEGs and PNGs that we humans will never notice. SVGs ~~can~~ should also be gzipped.

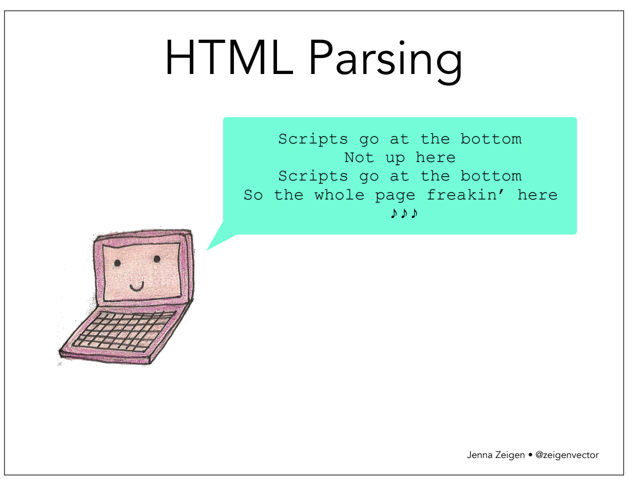
You also should to make sure you're using the right image type for the style of image. Vector graphics, i.e. SVGs, are all the rage these days, but they are best for simple, geometric illustrations like line drawings. Raster images, like JPEGs and PNGs, are generally bigger files



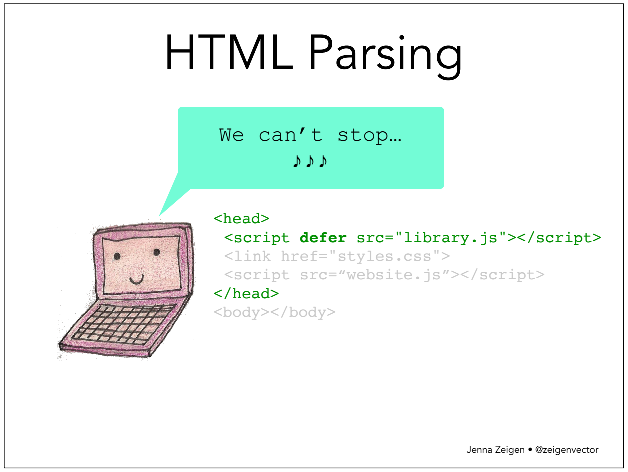

Use system fonts if you can! If it's not the case, chances are high that the web fonts you are serving include glyphs and extra features and weights that aren't being used. You can ask your type foundry to subset web fonts or subset them yourself (i.e. include only Latin with some special accent glyphs) if you are using open-source fonts to minimize their file sizes.



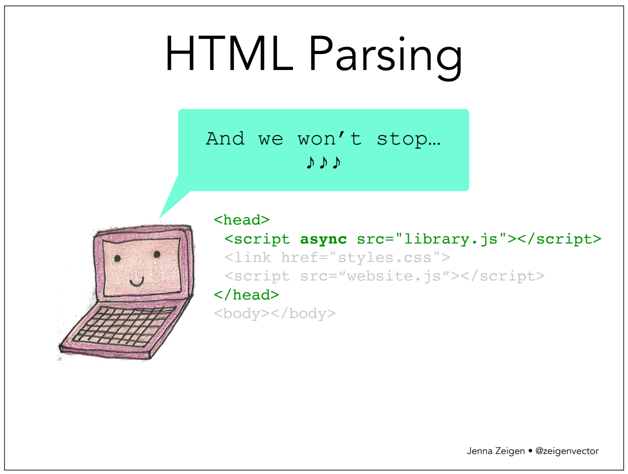

As the parser is doing its parsing and it runs into a JavaScript include, further parsing stops. This is because JavaScript has the ability to manipulate the DOM, using `document.write`, and any HTML that gets added must be fed back into the main parser. The parsing of the document halts until the script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



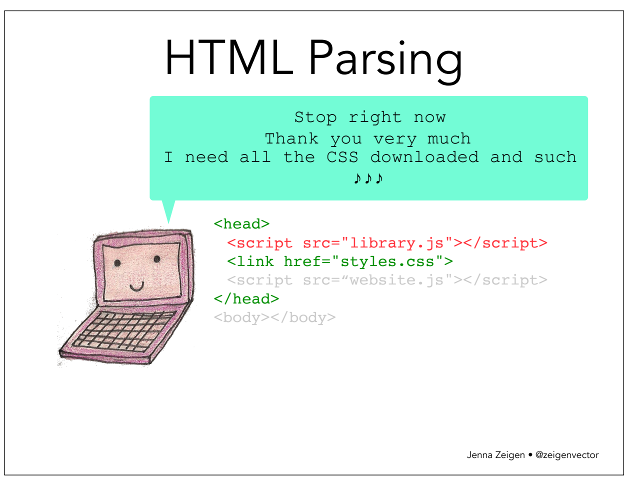

This explains the common optimization of putting scripts at the bottom of the page, right before `</body>`. Thus, when this blocking inevitably occurs, all the good things have already happened and you're well on your way to a render!




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.

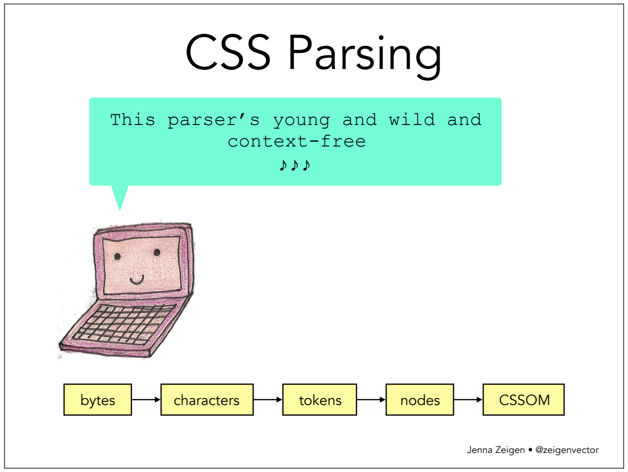



HTML5 adds an another option, to mark the script as asynchronous with the `async` attribute so it will be parsed and executed by a different thread, in parallel. With this, you're kinda promising you won't call `document.write`, or you're okay with it getting thrown out the window. (that was an attempt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

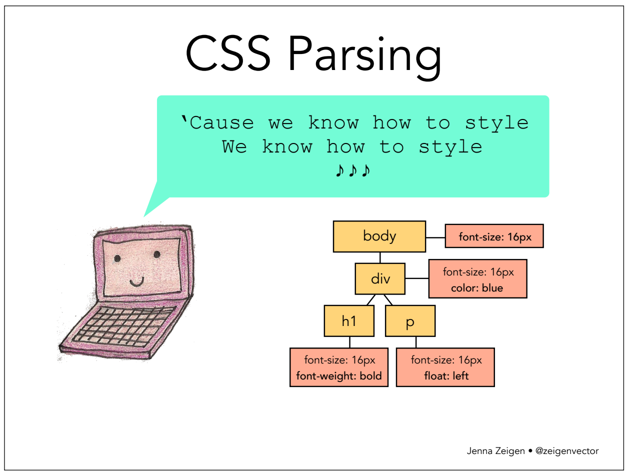



CSS also does its own kind of blocking. It might seem that since CSS doesn't change the DOM tree, there's no reason to wait for it and stop the document parsing. There is an issue, though, of scripts asking for style information during the document parsing stage. If the styles aren't loaded and parsed 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.

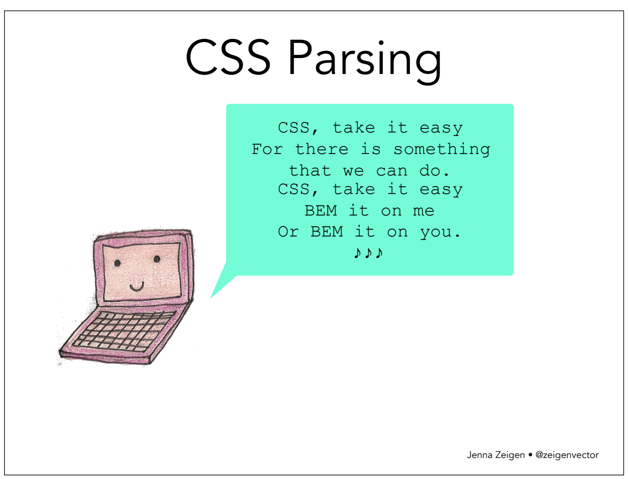
Different browsers have different ways of dealing with this blocking though. Firefox blocks all scripts when there is a stylesheet that is still being loaded and parsed. WebKit blocks scripts only when they try to access certain style properties that may be affected by unloaded stylesheets.



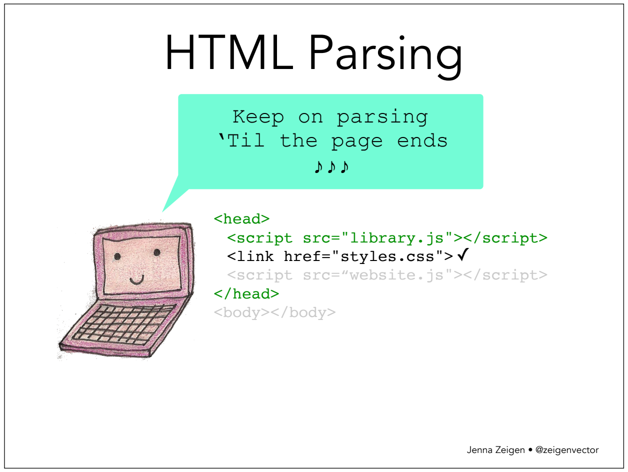

Just like with HTML, the browser needs to convert the CSS file into something it can work with. Instead of making a DOM tree though, it makes a CSSOM tree! The CSS parser, unlike the HTML parser, is context-free, though the steps in the process are largely the same.



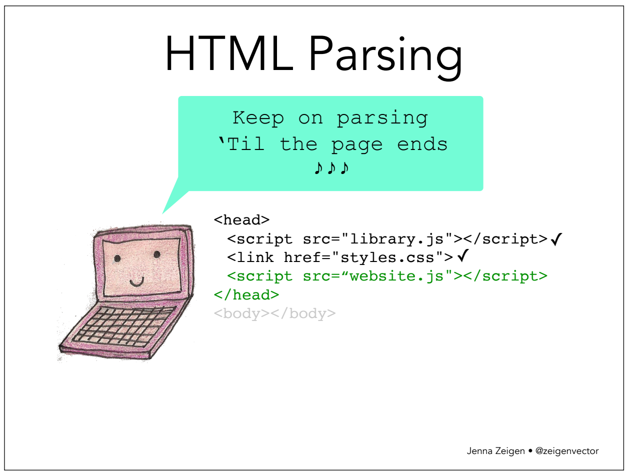

The tree data structure is well-suited for CSS because of the recursive nature of style application, the "cascade." It should be noted that the CSSOM tree only includes overrides to the browser's default stylesheet, so things that are included in the application's stylesheet and inlined in the HTML.



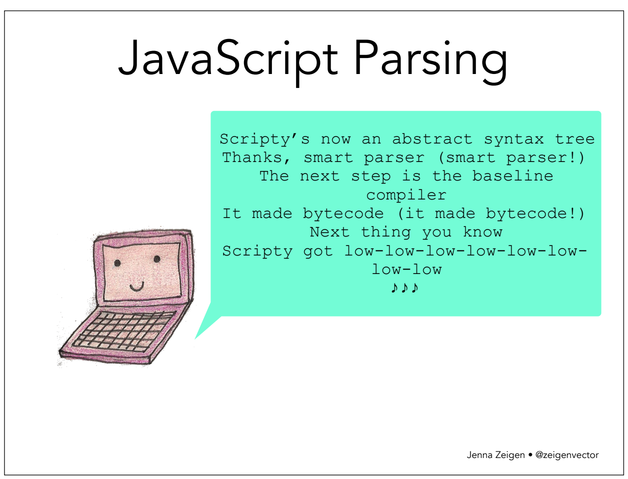

Reducing CSS selector complexity is one more step you can take to improve performance, and using a CSS methodology is one way to help control selector nesting. The "deeper" the styles go, the longer it takes the browser to figure out if there's a match for the rule it's currently looking at. Many of the popular methodologies rely on giving each element that need to be styled a single class based on strict naming rules so it's easy for the browser to figure out if a rule should be applied or not. Some of these methodologies are BEM, SMACSS, SUIT, and OOCSS.




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

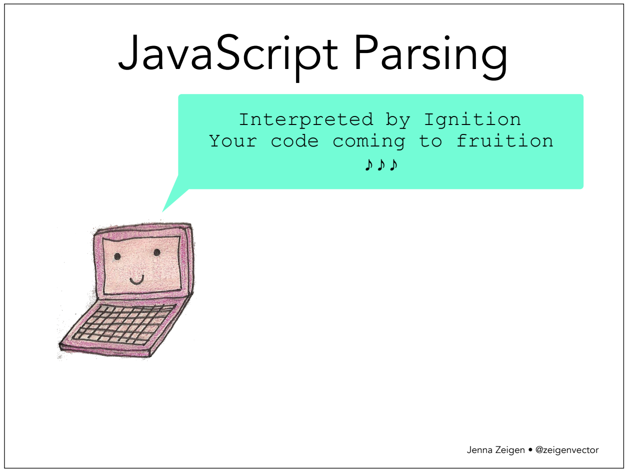



Successive assets then can be downloaded and parsed.

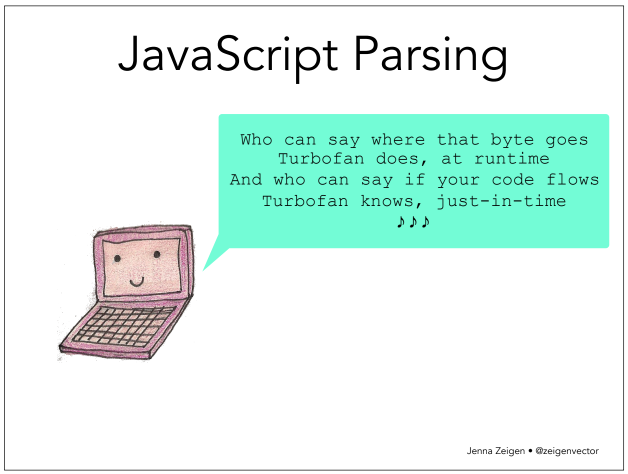



JavaScript is also context-free and can use a regular parser, but browsers these days make it more complicated in order to optimize the process. V8, the JavaScript engine in Chrome, uses two parsers—eager and lazy — to eventually create an abstract syntax tree (AST) and scope structure. The eager parser 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.

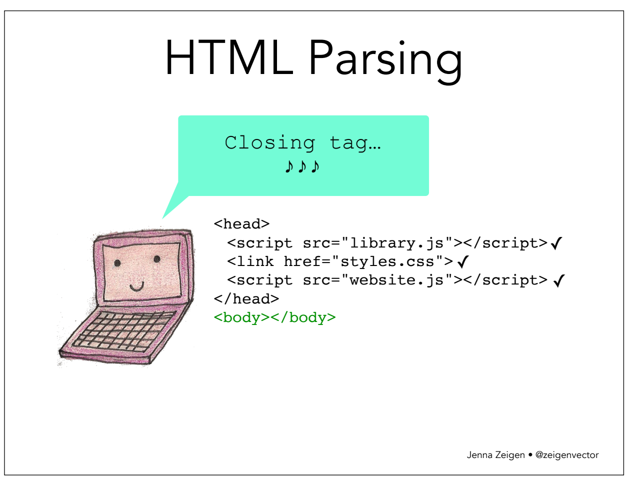
The AST and scope structures get fed into a baseline compiler and turned into low-level code. In the case of V8, a bytecode generator walks the AST and scope structure and turns it into



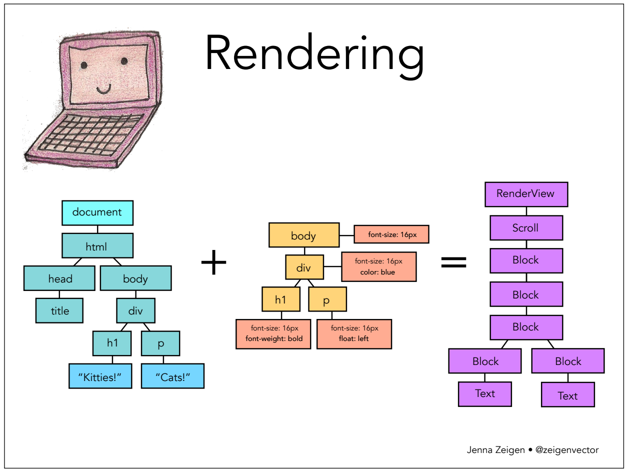

The low-level code then gets executed. In V8, the bytecode is then executed by the interpreter, called Ignition in V8



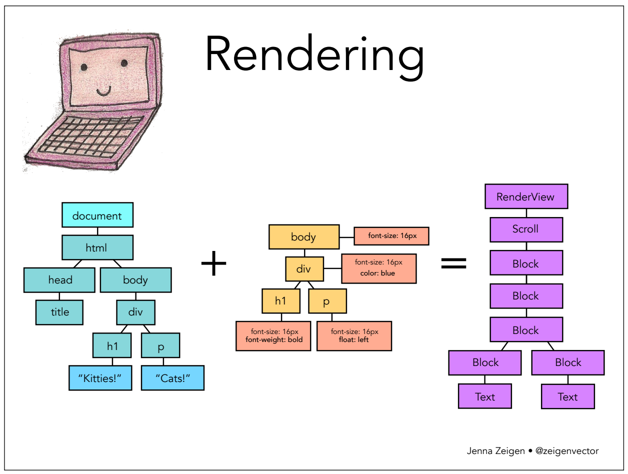

The bytecode also gets fed to the optimizing compiler, which recompile's functions that get used a lot and are worth speeding up. In V8, there is only one optimizing compiler, called Turbofan. Because JavaScript has a just-in-time (JIT) compiler, the code gets compiled at runtime and information used 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



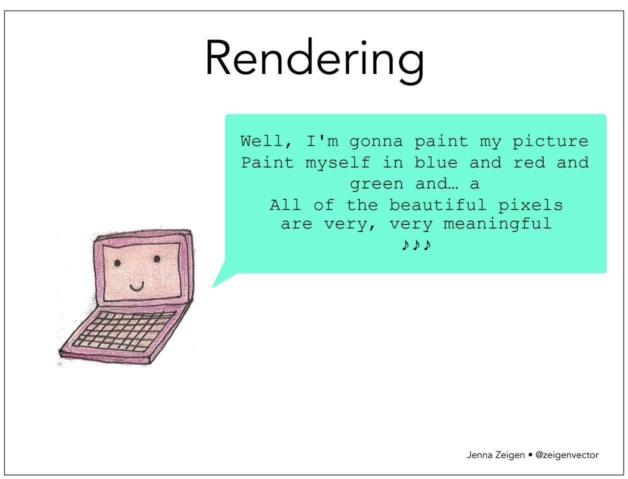

Eventually, the whole document will be fed into the parser and made into DOM tree.



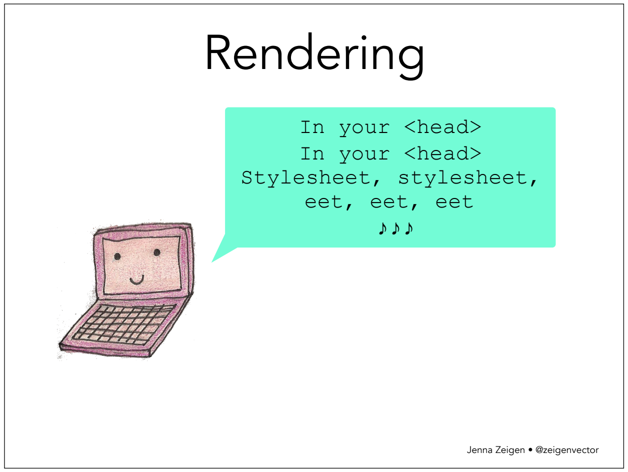

While it's important to download all the assets in the right order, rendering is just as integral a job for the browser. To kick off rendering the page, CSSOM and DOM trees are combined into a render tree, which is then used to compute the layout of each visible element and serves as an input to the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



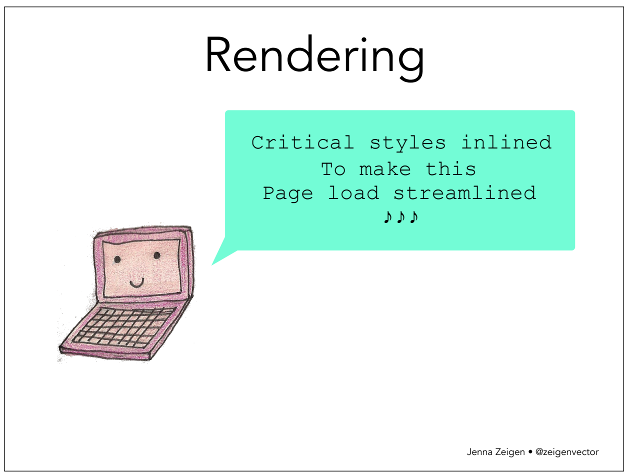

Because of this, HTML and CSS are both render-blocking resources, which means that the browser will hold rendering of any processed content until the DOM and CSSOM trees are constructed.



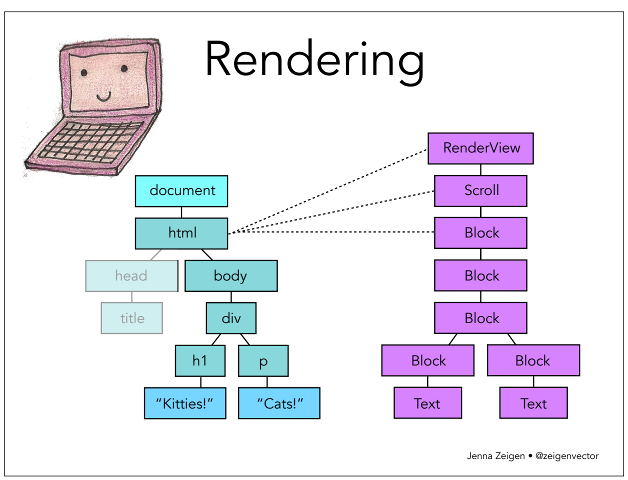

The idea behind a lot of the performance tricks is to allow us to create a render tree as soon as possible. This process has come to be known as "optimizing the critical rendering path". The metric that measures this has come to be called the "Time to first meaningful paint."



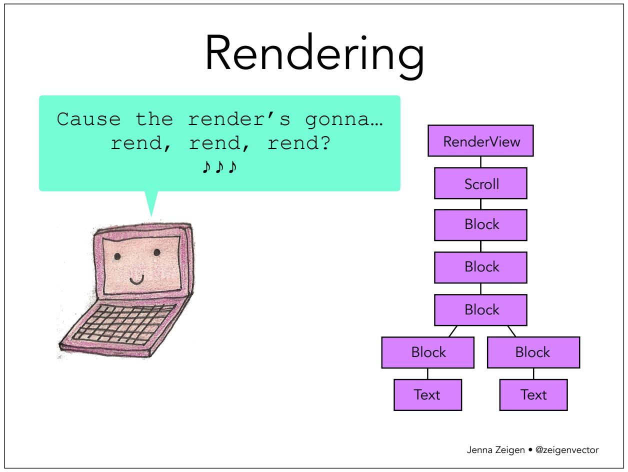

This is one reason why stylesheets are supposed to go at the top of your HTML document, in `<head>`. Putting stylesheets near the bottom of the document prohibits progressive rendering in many browsers. These browsers block rendering to avoid having to redraw elements of the page if their styles change. 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…




Within the past few years, it's becoming more common to inline critical CSS into the head of the document, and experts say that this will get you the best performance. Critical CSS is the minimum blocking set of CSS that we need to make the page appear to a user. The idea here is that that we should prioritize everything that will get the first screen's worth of content to the user, also called the "above the fold" content, within the first few TCP packets. By inlining the CSS, you are cutting out the time otherwise needed for a round-trip to grab a styles. Due to the limited size of packets exchanged that early in a TCP transaction, your critical CSS budget is around 14KB.

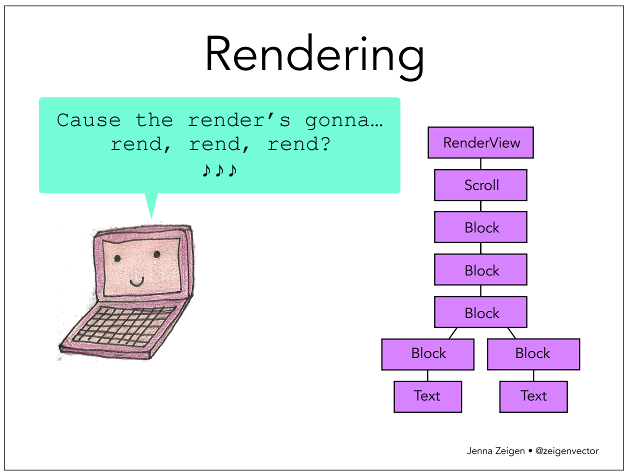



Elements in the render tree correspond to elements in the DOM tree, but it's not 1:1. The render tree only includes things to be rendered, so no non-visual DOM elements like `<script>` or elements with the style `display: none`. Elements with the style `visibility: hidden` will be in the render tree,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



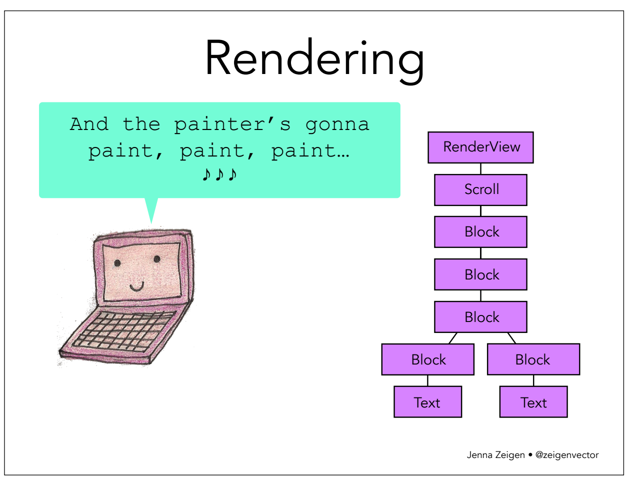

While the render tree contains information about what nodes are visible and computed styles, we haven't yet figured out where all these elements are supposed to go or how big they're supposed to be. This is all part of the "layout" process, also called "reflow."

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.



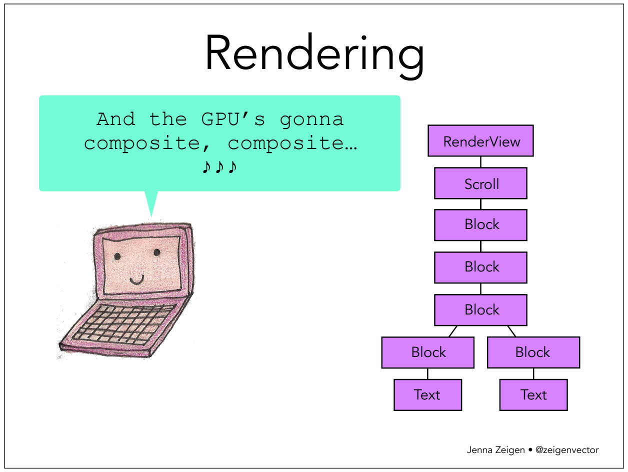

In order not to do a full layout for every small change, browsers use a "dirty bit" system. A renderer that is changed or added marks itself and its children as "dirty", which just means that it needs laying out. There are actually two flags: "dirty" and "children are dirty." The latter means that while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



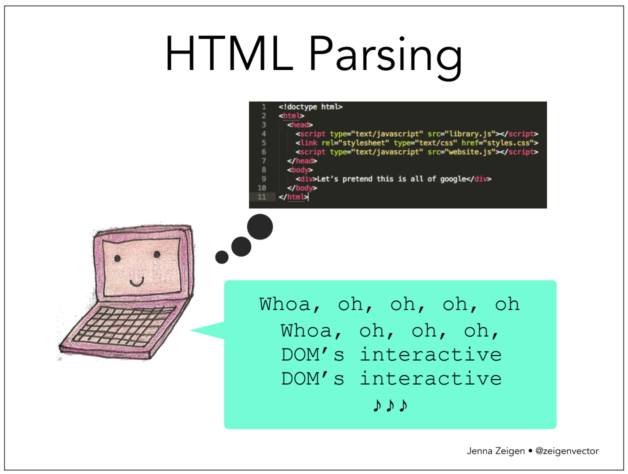

Once we know the location and dimensions of all the elements in the render tree, we can then start representing everything as pixels on the page — painting! In the painting stage, the render tree is traversed recursively and the renderer's "paint" method is called to display the content. This process 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!

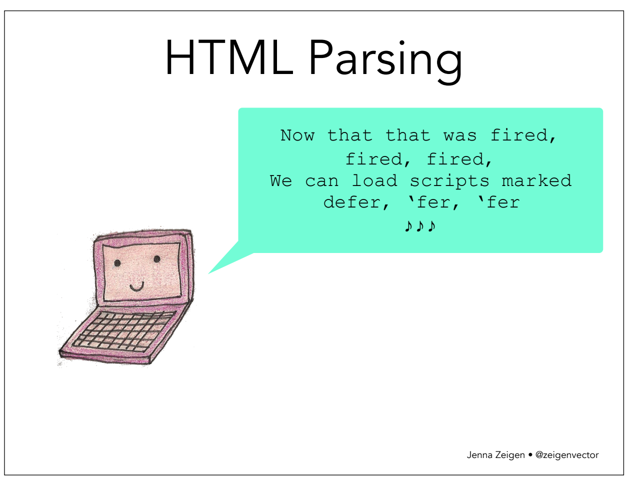



Then the bitmaps are handed to the GPU for compositing (or more technically: smushing all the layers together).

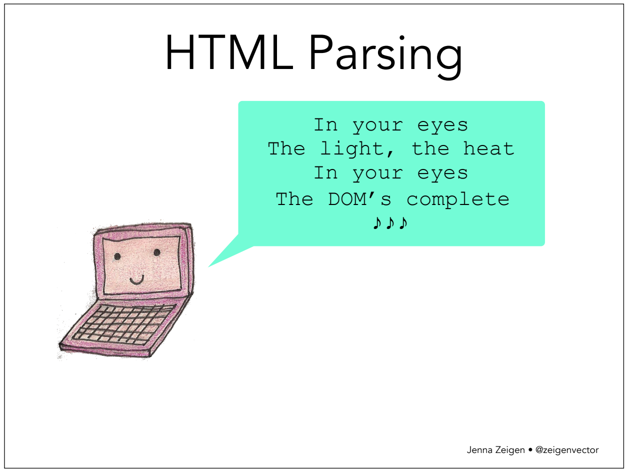
This whole process can occur at 60 frames per second, which happens to also be the frame rate of human vision, or "flicker fusion rate of cones". You might have also heard this as every 16.66 ms.



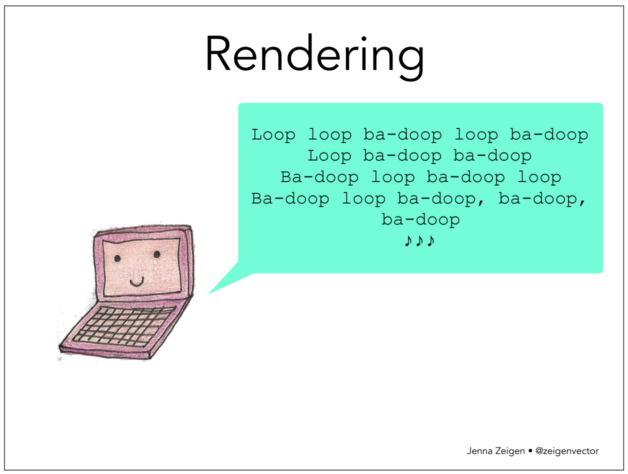

When the HTML document is fully downloaded and parsed, the page is marked as "`interactive`" and the `DOMContentLoaded` event is fired, without waiting for all the assets to finish loading. At this point, you can click around the page— it's interactive!




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

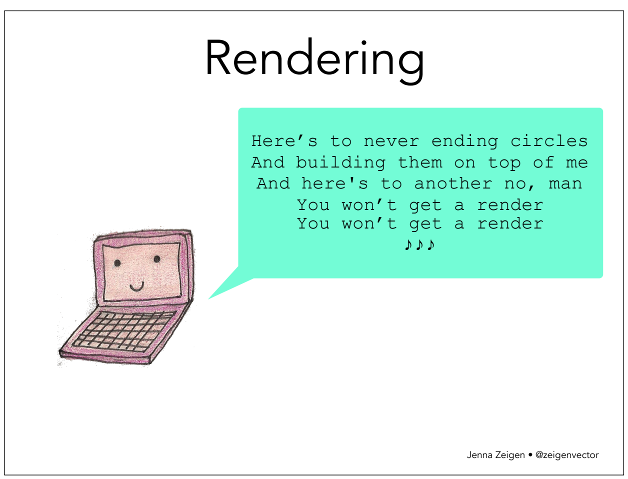



Finally, once every resource has been downloaded and parsed, the document is marked as complete, and the `load` event is fired.

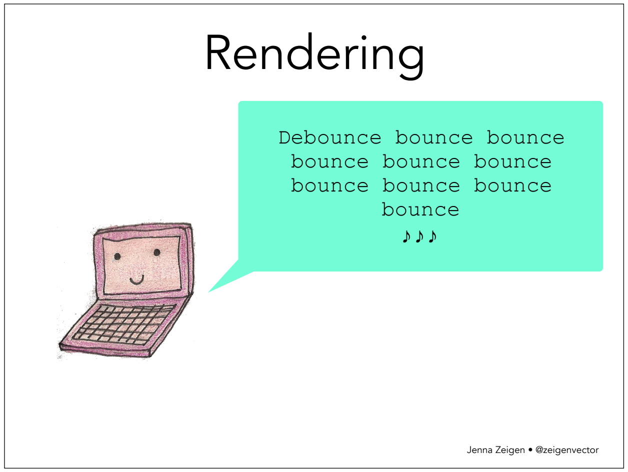



But websites these days are interactive. Users will click and drag and type, and pieces will change. And the browser must update based on these interactions.

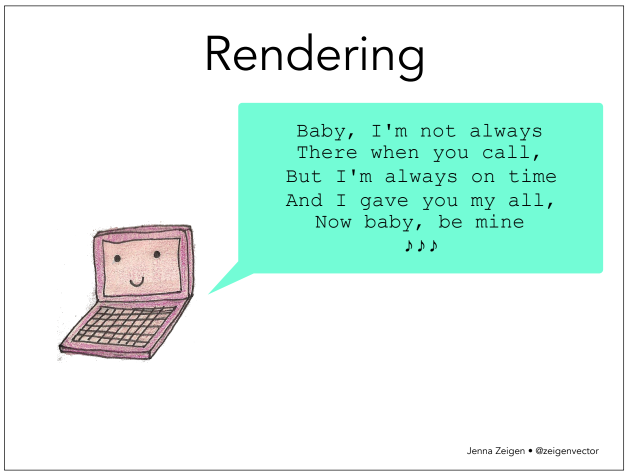
But renders are constrained by the speed of your JavaScript. JavaScript is single-threaded, running on the browser's main thread along with all the style, layout, and painting calculations. Everything that gets called in JavaScript goes onto the stack. Synchronous calls go right on, and asynchronous callbacks, like those from Web APIs, get thrown into a callback queue and are able to be moved to the stack by the event loop once the stack is cleared. There is also a render queue that tries to get renders done sixty times per second, but renders can't happen if there is anything in the JavaScript call stack.



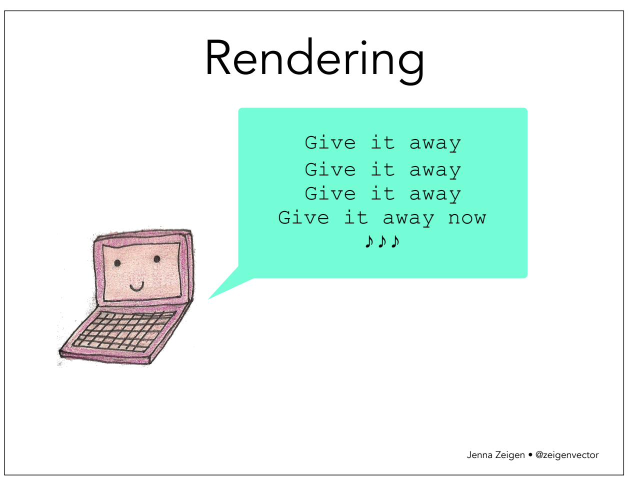

That's why nothing can render when you're in an infinite loop, because the call stays on the call stack, preventing renders from getting initiated.



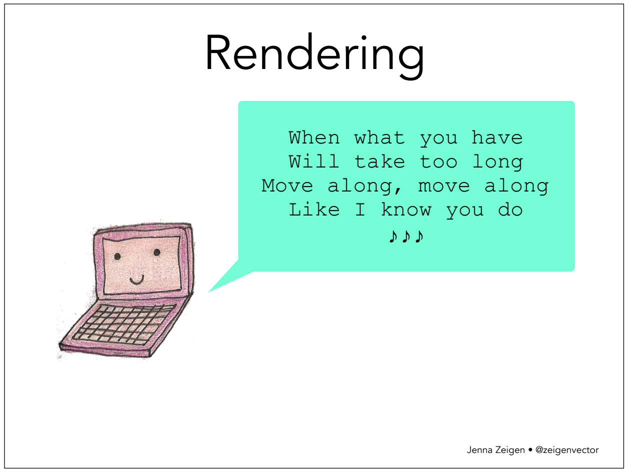

If you're effecting visual changes via JavaScript, you want to do it at just the right time. Wrapping your updating function in `window.requestAnimationFrame` tells the browser that you wish to perform an animation and requests that the browser call your provided function before the next repaint. This particularly applies to scroll handlers. Wrapping them in this function debounces them until the next time there will be a paint.



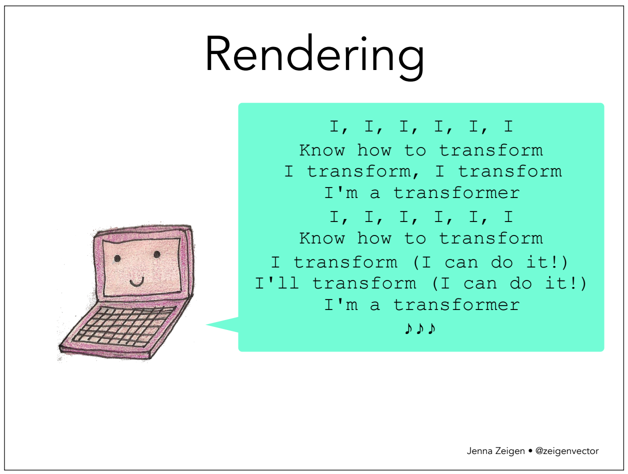

If the timing of updating where your pixels are on the screen is off, you might drop frames. For example, if your user is supposed to be perceiving motion, but updates aren't occurring sixty times per second, your users will be able to perceive that the motion wasn't smooth, that the animations are janky or laggy.




If you're also doing computationally intensive stuff with your JavaScript that isn't pushing pixels around, chances are you can move it to a Web Worker. Web workers are a way for you to run JavaScript in background threads. Web Workers are pretty versatile, but they do have some limitations. They don't have access to the DOM, so you cannot do any DOM manipulation within them. They also don't have access to some of the things that live on the window object, but you do have access to some, like the Cache, WebSockets, XMLHttpRequest and fetch, and even WebGL.

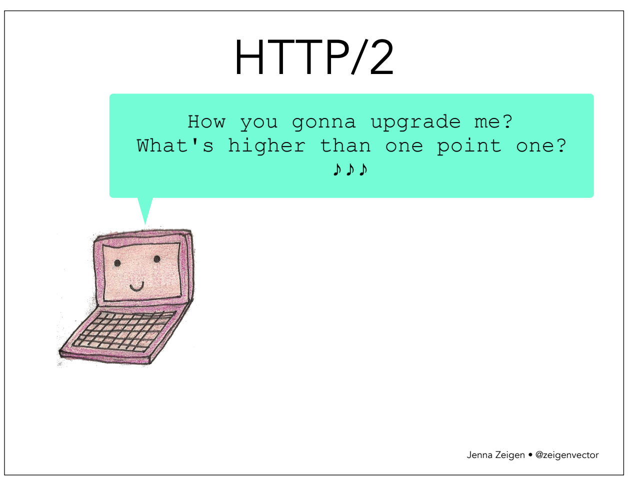



As I mentioned, any JavaScript computations on the browser's main thread have to complete within 16ms to ensure an unblocked next paint. Moving things that might take longer of the main thread so they aren't gumming up the crucial call stack allows this to better happen.




If you need to move something or make it disappear, use transform and opacity, as opposed to position and display. This is because these properties can be handled by the compositor alone. Compositing, the last step in rendering, does not require a re-layout or a re-paint, which are expensive operations. On the other hand, positioning and display, require a re-layout, and subsequently a re-paint and re-composite.

The caveat here is that the element which these properties are changed on need to be in its own compositor layer. You can do this by adding `will-change: transform` to a moving element's list of properties. Use this magic sparingly, because there is a memory hit from maintaining each layer, which is particularly pricey on devices with limited memory such as




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 the of the optimizations discussed will be obsolete. Features include:

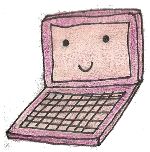
- HTTP header compression (no repeated headers being passed back and forth)
- Multiplexing (Allows the browser to fire off several requests at the same time on the same connection and receive responses asynchronously)
- Server push technologies (sends assets with the HTML, because it knows you're gonna need it)
- Loading page elements in parallel over a single TCP connection

# 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



# Resources



General
- What Happens When (Github)

Transport
- DNS (Wikipedia)
- UDP (Wikipedia)
- TCP (Wikipedia)
- TLS (Wikipedia)

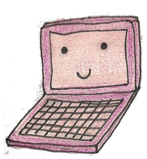
Parsing, Rendering, and Painting
- How Browsers Work (HTML5 Rocks)
- Marja Hölttä: Parsing JavaScript - better lazy than eager? (JSConf EU 2017)
- Franziska Hinkelmann: JavaScript engines - how do they even? (JSConf EU 2017)
- Critical Rendering Path (Google Developers)
- True Grit: Debugging CSS & Render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



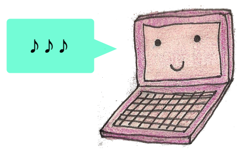
# Resources

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)

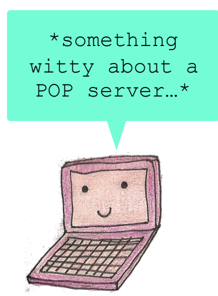


**Complete Tracklist**
Carly Rae Jepsen - Call Me Maybe
Radiohead - Creep
Taylor Swift - Blank Space
Taylor Swift - We Are Never Ever Getting Back Together
Taylor Swift - Wildest Dreams
Sarah McLachlan - I Will Remember You
Aretha Franklin - R-E-S-P-E-C-T
Alanis Morrisette - Hand In My Pocket
The Supremes & The Four Tops - The Bigger You Love, The Harder You Fall
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
The Beatles - You Know My Name (Look Up The Number)
Calvin Harris ft. Ellie Goulding - Outside



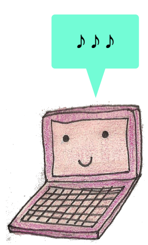

Jenna Zeigen • @zeigenvector

**Complete Tracklist (continued)**
Jimmy Eat World - Sweetness
The Notorious B.I.G. – Sky's The Limit
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
Daft Punk – Harder, Better, Faster, Stronger
Taylor Swift - Blank Space (yes, again…)
Cher - Believe
OutKast - Hey Ya
Blink 182 - All The Small Things
Sia - Cheap Thrills
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



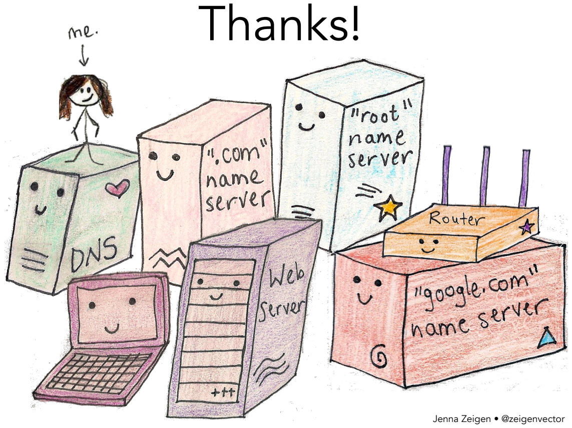

Jenna Zeigen • @zeigenvector

**Complete Tracklist (continued)**
Mika - Relax, Take it Easy
Britney Spears - Till the World Ends
Flo Rida - Low
R. Kelly - Ignition
Enya - Only Time
Semisonic - Closing Time
Counting Crows - Mr. Jones
The Cranberries - Zombie
Taking Back Sunday - Cute Without the E
Taylor Swift - Shake It Off
Imagine Dragons - Radioactive
Ellie Goulding - Burn
Peter Gabriel - In Your Eyes
Salt-N-Pepa - Shoop
CHVRCHES - Never Ending Circles
R. Kelly - Ignition (yeah, again)
Ja Rule ft. Ashanti - On Time
Red Hot Chili Peppers - Give it Away
The All-American Rejects - Move Along
Gnarls Barkley - Transformer
Beyoncé - Upgrade U




Jenna Zeigen • @zeigenvector

# Thanks!




Jenna Zeigen • @zeigenvector