What Happens When You Type In a URL & Hit Enter

Way more detail than you could possibly ever want, guaranteed!

Presented by Nick Evans for ECT, June 2024

Why the Question?

- Touches EVERYTHING in Tech
  - No matter your role - some (probably most) parts of this impact your job day-to-day
  - Applicable if you're service desk, web dev, analyst, PM, telecom tech, or even a cabling contractor

- Great for interviews
  - Open-ended
  - Can go in a lot of directions
  - Which part(s) people focus on can be interesting 😏

- Kind of a fun deck: lots of gotchas and little details you don't generally think about!

What Happens When You Type In a URL & Hit Enter

Well, what kind of keyboard are you using?

- You're *probably* using a USB keyboard, so let's assume that for the sake of brevity.
  - **BUT:** works a bit differently with an olde-style PS2 keyboard!
  - Directly sends interrupt requests to the processor on IRQ1

- But before we get into how USB keyboards work, consider the physical device!

- Keycap -> switch -> matrix circuit doing "the magics"
  - Switches are a whole thing on their own: membrane vs. mechanical
  - Inside mechanical switches, there's linear vs. tactile
  - People optimize their switch type for task -- e.g. linear offers the shortest travel time (at the cost of no tactile feedback)

USB Keyboard: Crash Course

- USB spec defines standard for the most common HIDs: keyboards, mice, and game pads
- Plugging your keyboard in causes an intense negotiation to register it w/ the USB host & potentially ask it for more electricity
  - To power your RBG, of course
- Pressing a key has distinct states: key down, key up
- Keys have integer key codes
- When you press something, the keyboard's own hardware registers it & stores that in its own buffer
  - Key down 13, key up 13 (the enter key!)
- USB host controller polls keyboard every ***X*** ms
  - Polling period set up during negotiation!

Key press Data Getting to Software

- USB host controller gets the data out of the keyboard’s buffer

- Hands raw data off to the USB HID Keyboard driver (which is generic)

- USB HID driver passes keypress data to the OS’ hardware abstraction layer

- Of course, for non-USB keyboards, some different stuff happens!

- But it all eventually ends up hitting the OS’ hardware abstraction layer
  - Which can then hand the keypress events off to its software
  - So devs don’t have to care what kind of keyboard it is!

De-rail: Keyboard Hardware Limitations

- Physical keyboards occasionally are bad at detecting keypresses when you're mashing lots of buttons!

- This usually happens with cheaper keyboards: the circuit isn't really designed with more than 1 or 2 keys being pressed at once in mind
- So the keyboard matrix circuit may mis-detect a combination of 3 keys as 4 keys
  - And then another anti-ghosting measure will detect *that* and drop the "ghost" 4th key press
- Further fun: not every circuit behaves the same, so you end up with different combinations causing problems across different models!

- Higher-end keyboards feature 3-key (or N-key) rollover
  - Basically, they spent more money on their keyboard matrix circuit and it scans for key presses on each individual button
- Microsoft has a demo of the ghosting & N-key rollover detecting stuff

De-rail: Other Types of Keyboards

- PS2 & USB keyboards aren’t the only types available though.

- Perhaps the most common in 2021: virtual keyboards!
    - Cuz, you know, iOS/Android

- Apple touch screens feature capacitive layer that detect electrical changes where you put your fingers
    - Keydown / keyup events are detected in software by the screen reporting touch event ***X***, ***Y*** coordinates
- Much more reliant on software (obviously!)

Cool, Key Pressed!

We’re seven slides in
and all we’ve talked about is keyboards 😀

OS Routes the Keyboard Events

- Oh, did you think we were done talking about keys?
  - We’ve got keyboard input into the OS
  - But it still needs to find its way into a userland application like Chrome

- OS is aware of which application has focus
  - Differs a bit between Windows/UNIX-likes
  - Win32K (the kernel) is responsible for giving focused userland app the keyboard events
  - On Linux, the kernel just barfs keypresses and a userland GUI server like X.org is the intermediary between the kernel & application
    - MacOS is similar to Linux

Application Receives a Key Press

- Applications receive keypresses as events and generally have event listeners bound
- When you focus a text input field like the URL bar, your GUI API generally helps you out and just fills it in as people type
- But apps can also listen for specific events w/out an input field
  - e.g. ctrl+r = reload the page

- Let’s assume “google.com” has gone into Chrome’s URL bar
- <u>but</u> we **haven’t** hit enter yet!

The Chrome URL Bar

- The term “URL bar” isn’t a very good description, ‘cuz it does a couple things when you type into it:

  1. Check your history
  2. Check preferred search engine for potential hits & related search terms
  3. Look stuff up in Wikipedia & other fact-giving services

- That’s all before even hitting *<enter>*

- Checking your history consults a sqlite database Chrome has in your profile
  - It factors in a couple things, including how recently you’ve gone to that site

- But the suggested search terms & looking up celebrity photos from Wikipedia/etc goes over the network to Google’s APIs

Networking, Part 1

This is going to be dense, since I have to cram 60 years of incredibly boring history into a slide deck.

Might wanna grab another slice of pizza at this point ...

Over The Network And Through The Woods To G's House We Go

- Talking to the network is an involved process

- For brevity, we'll assume the network is already set up and is able to talk to the internet
  - So we won't go into stuff about IP assignment via DHCP or otherwise

- Your computer has a gateway to the rest of the internet, and knows to route traffic over to that gateway

- But we're getting ahead of ourselves!

- Chrome is going to want to hit some <*google.com*> URL to get the suggestion data
  - But <*google.com*> isn't an IP that your computer can route to, so you have to do a DNS lookup.

Welcome to DNS

- We invented DNS to turn nice domain names into IP addresses after the internet got too big to remember everything
- It’s a distributed fault-tolerant system
  - Which often is at fault for outages, ironically
  - But this isn’t an ops slide deck so we won’t make too many jokes at DNS’ expense
- DNS servers get requests from clients: “resolve google.com please”
- Your closest DNS server (probably run by your ISP) isn’t in charge of <google.com>, so it has to figure out who *is* in charge of that.
- So DNS servers ask a chain of increasingly-important servers what IP google.com resolves to

Resolving a Domain Name

- Your system's configured resolver (generally called a *recursive resolver*) will ask a root nameserver about <*.com*>'s authoratative nameserver.
  - The root nameservers are the 13 "servers" that make the whole internets work
- Root nameserver tells you to ask some IP about <*.com*> domains

- Recursive resolver then asks the <*.com*> server what server is in charge of <*google.com*>

- The recursive resolver can then ask it for the IP that <*google.com*> has

- If you had more subdomains, this could continue for more levels
  - *E.g. <api.chrome.google.com>*

DNS Records

- Of course, DNS is more complicated than “what’s google.com’s IP address?”

- DNS records you’d be looking for here are A & AAAA
  - But if you had <*chrome.google.com*>, you could be looking for A, AAAA, CNAME, or ALIAS.

- The A & AAAA will give you IPv4 & IPv6 addresses, respectively.

- CNAMEs and ALIASes are pointers to other records: <*chrome.google.com*> could have an A record for 8.8.8.8, or a CNAME pointing you to <*browser.google.com*>
  - So CNAME = more recursion!

Down The Rabbit Hole: Networking

- We’ve already used the IP protocol to make DNS requests, but I haven’t explained what’s happening there!

- The IP protocol is a means of addressing packets of data to other networks, using IP addresses.

- IP protocol works irrespective of the physical transport (ethernet, fiber, etc) & irrespective of what the packets of data contain (e.g. Netflix or timesheets)

- IP is essentially “the internet” - - this is a logical layer that everything talks to each other with

- IPv4 and IPv6 serve the same purpose, but use different addressing schemes
  - IPv4 is limited to 4.2b IPs, since they used a 32-bit integer for the IP address field in the spec
  - But that’s NOT ENOUGH
  - IPv6 uses a 128-bit integer, which is enough IPs for every single molecule in the universe

But How Does My Computer Actually Move IP Packets?

- That depends!

- The other “layers” above & below IP are meant to be modular
  - So you can send IP packets over dial-up or fiber optic cable

- Generally, your computer will have an Ethernet card with a copper wire hooked up to a router.
  - Wifi will pretend to work the same, except with radios instead of wires. And some additional error-correction done below the IP protocol.

- When you want to send a packet to Google via your router, your computer will hand it off to your router’s IP
  - So your Ethernet card has to make an ARP request on the network, asking if anyone knows the MAC address for your router’s IP
  - E.g. “hey what’s the ethernet card I’m connected to that has 192.168.0.1 as its IP?”
  - And the router will see this shouted out onto the network and say “It’s me! Send it to 00:0a:95:9d:68:16!”

But How Does Ethernet Actually Move Packets?

- At this level, they aren't considered "packets". That's a concept for the IP protcol.
  - We call 'em frames down here, and they can be sliced up as the hardware requires
- But ethernet uses electrical signals over copper!
  - Fiber will use light through glass
  - WiFi will use extremely shrill noises over air
- The hardware will address other stuff physically connected to it (e.g. MAC addressing for Ethernet)
- And translate some kind of signal into data before passing it back up the stack (or vise-versa)

OK, but How Do I Know My Packets Got There?

- Above IP, we can have the TCP protocol
  - TCP is all about transmission control
  - You shake hands and acknowledge receiving data
  - So Chrome knows its packets sent to Google's servers are being received!

- But we got here from DNS, so ignore everything I just said.

- DNS requests don't (typically) use TCP -- it uses UDP instead!

- TCP adds a whole bunch of overhead for what is otherwise a very small bit of data: "what is the IP for <*x*>" / "the IP is <*x*>"

- That response is acknowledgement that the request was received, so that part of TCP is useless too

OK, Well, How Does My Computer Deal With TCP or UDP?

- Your kernel has a network stack that knows about UDP & TCP, IP, ARP, and whatever you need for your hardware.

- Developers generally don't have to care about this, beyond indicating if they want TCP or UDP when they open a socket

- "*Wait,*" you say, "*what's a socket?*"
  - A socket is an open connection to an IP address on a port using a transport layer protocol
  - It can send data back & forth between the two computers

- "Port?"
  - It's an apartment number to the IP's address.
  - A server will let its various services (website, DNS, email) listen to a port
  - So when you connect to port 53, you're *probably* talking to a DNS server

Making the API Request to Google

- Getting back to the topic at hand…

- Once your recursive resolver gets the IP address, it will cache it for some time

- And now Chrome can ask Google if there are any suggested search terms, good search results, or pictures of Nicholas Cage to display as potential options!

- This API request happens over HTTPS
  - Chrome builds a URL, e.g. https://suggestions.google.com?k=<*what you typed*>

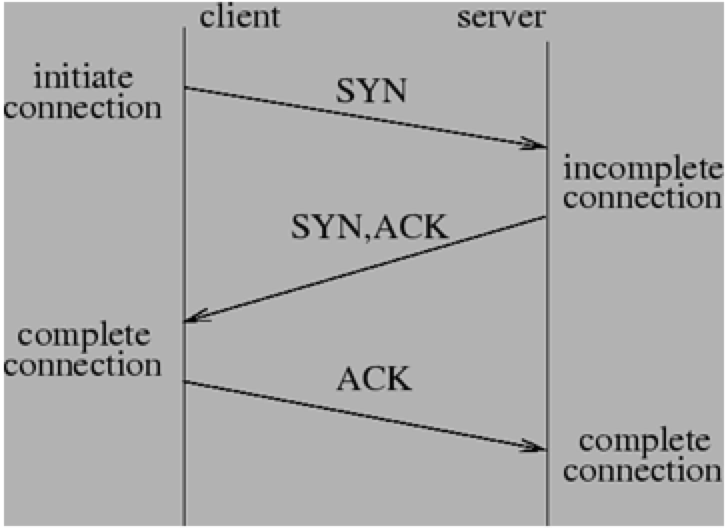
- Chrome opens a TCP socket to the IP on port 443

Revisiting TCP

- We brushed off TCP earlier since we were worried about DNS requests over UDP

- But a TCP request is a much more involved process than “send a packet, hope it arrives”

- Everything requires confirmation & confirmation of getting the confirmation
  - Opening the socket
  - Each packet being sent
  - Closing the socket



TCP is Noisy

- With all the back & forth acknowledgments, TCP requires a lot more bandwidth
  - And if something isn't acknowledged properly, it will try to re-send the packet

- TCP packets also contain information about their order
  - So the receiver can get them out-of-order and hold them in a buffer until it's got the missing packets

- The tradeoff between TCP and UDP is reliability: you can guarantee stuff is getting there in the right order with TCP

- Which makes it great for HTTP, since you're sending so much more data than "what is <*x*>" / "it is <*x*>"

Setting Up The HTTPS Secure Tunnel

- HTTPS is the good-ol' HTTP protocol sent through an encrypted TLS tunnel

- Setting up the TLS tunnel happens first

- TLS guarantees two things:
  - The connection is free from eavesdroppers (**encryption**)
  - The server belongs to whomever the certificate says it does (**trust**)

- The server will proffer a certificate with information about who issued it, what website they issued it for, when they issued it, and how long it's valid for

- Browser checks all of this info
  - Current time is between issued at & expiration dates
  - Certificate is for the domain name Chrome is talking to
  - Certificate was issued by a trusted Certificate Authority

Certificate Authorities

- Chrome has a list of Certificate Authority certs that it trusts
  - This list is maintained by a consortium: Google, Mozilla, Apple, Microsoft, etc all sit on the board

- These CA certs sign a website’s cert
  - The CAs are *supposed to* verify that the person paying them $15 for a cert on northwestern.edu is actually part of Northwestern University
  - This is how trust is established!

- The website’s cert may not be signed directly by a trusted CA key

- Many CAs use one (or more) “intermediate” certificates that they can rotate out more easily than the trusted CA cert that *every single browser bundles & ships to millions of computers*

- Webserver may need to help the client out by proffering the whole chain of intermediate certificates

De-Rail: Name-Based Virtual Hosting & TLS Certificates

- Name-based virtual hosting is the practice of having one webserver on one IP serve several websites

- The server knows which site to give you based on the HTTP *Host: google.com* header

- But the TLS tunnel is established *before* the HTTP request is made

- So the TLS folks came up with the Server Name Identification (SNI) extension to TLS

- That *Host: google.com* header becomes part of the TLS handshake, so the webserver can look up the right certificate to proffer

- This is an OPTIONAL extention to TLS
  - It's widely supported now, but real old WinXP machines may have issues

Revoking Certificates

- A website’s certificate consists of two parts:
  - Public key: contains all the data we’ve been talking about)
  - Private key: used to encrypt data

- The private key is intended to remain private. If it is leaked, somebody could decrypt the traffic and use it for evil!

- Accidents (and Heartbleed) happen, so there is a way for a CA to revoke a certificate
  - Actually several ways…

- When a browser is proffered a certificate, it will check two things to see if it’s been revoked:
  - Certificate Revocation List
  - OCSP Status

Certificate Revocation Methods

- Certificate Revocation Lists are a big list of certificates that have been revoked
- Browser downloads this list every so often, and then it can check it (ON YOUR COMPUTER) whenever it needs to
- Everyone downloads the full list

- OCSP is more like an API that you ask about a specific certificate
- Advantage over CRL is that the OCSP status is *always* correct, whereas your CRL may not have been updated in the last few hours
- Disadvantage is that somebody may have a record of what specific site you’re visiting

Certificate Revocation Methods: Part 2

- Not every client uses CRLs or OCSP

- But browsers all use *at least* one method, if not both.

- Certificate revocation may not be in sync between the CRL and OCSP immediately

- The long and short is:

  Troubleshooting a “bad cert” when it’s been revoked is ***absolutely maddening***

Checking in on DNS

- That privacy concern from OCSP a few slides back is a good segue!

- DNS is not using TLS. It was invented long before we had enough processing power for encryption.

- This is bad for two reasons:

1. How can I trust DNS responses aren’t being tampered with by my ISP?

1. How do I know I can trust the DNS server?

DNS Security Features

- DNSSEC is an optional extension for DNS that adds public/private key encryption to records.

- An additional record that your recursive resolver can ask for (*DNSKEY*) is available
  - This is a public key you can use to validate a hash that comes back with your A/AAAA/etc records

- Every server in the chain that your recursive resolver follows must be signed
  - This is similar to the CA cert signing a website’s cert

- But there’s no inherent encryption
  - Chrome & Firefox are experimenting with “DNS over HTTP” so they can take advantage of TLS

The HTTP Protocol

We’re 32 slides in and finally getting to an *HTTP GET*.

Anatomy of an HTTP Request

```
HTTP GET /suggest?v=google.com HTTP/1.1
Host: google.com
User-Agent: ChromeLongStringBlabla 77 (Mozilla/IE/LOLCATS)
Accept: */*


{no body}
```

Anatomy of an HTTP Response

```
HTTP/1.1 200

Content-Type: application/json

Expires: Wed, 25 Jan 2021 15:15:22 GMT

Content-Length: 220


{“suggestions”: [{url: “google.com”, …}]}
```

Compression

- HTTP supports negotiation via headers

- Client can specify Accept-Encoding: gzip (or a couple other values)

- Server will compress the response & include a Content-Encoding: gzip header to indicate that it supported the compression requested

- Compressing the response saves on bandwidth, at the cost of a small amount of CPU client-side

HELLO, TCP

Remember we said TCP is noisy and unnecessary for DNS?

Well, we use it for HTTP because the packets that make up a request/response need to be in order!

Chrome Gets Suggestions Back & Renders Them

- API call complete, Chrome now has the data it needs to build the suggestions!

- Parse the JSON from the API call, merge it together w/ history data

- We can finally hit <*enter*>!
  - For brevity, let’s assume we didn’t pick one of the suggestions

Chrome Evaluates the “URL”

- Again: “URL bar” is a misnomer

- Chrome has to check to see if you’ve entered a valid URL
  - Otherwise it will send you to Google’s search results

- “URL” evaluation is fairly involved

- Does it have a protocol?
  - If not, assume https
  - If it does, is that protocol supported? E.g. *ftp://* was dropped in Chrome 82

- So our <*google.com*> should turn into <*https://google.com*>

Safe Browsing Checks

- Chrome (and most other browsers) have a “safe browsing” list
- Offers the Safe Browsing Update List API
  - Browser sends a hash on a partial URL
  - Response indicates if it’s evil
- Privacy implications:
  - You tell Google every site you visit

- Google maintains this list, but there are alternatives
- When the URL is determined to be **Evil:**
  - Chrome shows a warning screen instead of proceeding w/ HTTP request
- But, <*google.com*> won’t trigger a warning.

Now We Repeat Stuff

The DNS lookup, TCP/IP connection, TLS negotiation, revocation checks, and HTTP request/response all take place again.

This time, for your URL instead of the suggestion API.

De-Rail: International Domains

- DNS requires names to be ASCII
  - aka: the English alphabet
- But not everyone speaks a language that works w/ our alphabet
- Option 1: update every DNS server to support unicode hostnames
  - Internet says: no

- Option 2: punycode!
- This is a way for unicode characters to be represented as ASCII in DNS
- <*Bücher.example*> - must be punycoded!
  - Prefix punycode with *xn--*
  - Thus: <*xn--bcher-kva.example*>

De-Rail: The Cache

- I’ve assumed that Chrome *doesn’t* have the google homepage cached.
- Before the browser makes a connection, it will consult its local cache to see if it has that page already & can skip all the network stuff.
- Depending on headers on original response, browser may need to re-validate cached page before it can use it
- May send an HTTP OPTIONS request to check the Last-Modified or ETag headers
- These headers can be used by the server to indicate if the page has changed
- If it hasn’t changed: use cached copy!

Between Request & Response

We looked at an HTTP request and an HTTP response.

But what happens between them?

Between the Request & Response

- Short answer:
  - A webserver (eventually) gets the request
  - Builds a response: static files, or code runs
  - And then it builds the response

- Actual answer is complicated and varied
  - There are many, many, many ways to serve web traffic.

- In the case of the Google homepage…
  - The domain or IP are *probably* being served by your closest datacenter
  - *Probably* hits a load balancer
  - And the homepage of Google.com *probably* comes from a cache on their side
    - Partially: your login info is still there!

Better Example (For Our Purposes)

- A better example may be looking at something more familiar!
  - How about a request served from NU's AWS account by Lambda?
- Traffic gets to AWS over the internet
  - Hits the API Gateway in a region
  - Or hits CloudFront and is carried via optimized network paths back to the region, for Edge-Optimized API Gateways

- API Gateway unpacks the HTTP request & converts it into an event
- Is configured to invoke a Lambda
- Lambda spins up, looks at the event, runs code, builds a string response, and feeds it back through the chain to the client

A More Traditional Example

- Or in the case of NU's on-prem stuff, which is largely Apache …

- Request comes in to the server (maybe through a LB)

- Apache looks at the Host header (or SNI info), finds the right virtual host config

- Might serve up a static asset
  - May hand off to PHP-FPM or mod_php to build a response

- Might even go through the ForgeRock SSO module for Apache before processing the request
  - Which is looking at cookies & making more requests to back-end SSO servers

Rendering a Webpage

We're getting there!

Rendering the Document

- Chrome looks at the *Content-Type* header in the response to figure out what the document even is
- *text/html* is what we're mainly concerned with
  - But it can deal with other things, like *application/pdf* or *image/png*
- HTML gets parsed, turned into a DOM, and rendered
- HTML can contain references to other resources
  - Javascript files
  - Fonts
  - Images
  - Stylesheets
  - Videos
  - Etc etc etc

Fetch Other Resources

- Chrome will try to fetch linked resources as it encounters them

- Before it grabs something, it may need to check CORS to figure out if it *should* load the resource

- Caching rules apply here (and are usually more important here)

- JS will be fetched & run when it is encountered in the DOM
  - This means your JS can run before the HTML is fully processed & rendered!

- Same is true for CSS, but those rules will apply to any element matching their selector
  - Elements loaded or soon-to-be loaded!

De-Rail: Cross Origin Resource Sharing

- CORS is a security mechanism implemented **by browsers** to stop malicious folks from making HTTP requests “as your browser”

- If *<evil-nick.com>* could make a request through your browser, with your login info, to *<amazon.com/order-a-car>* ...

- So browsers restrict requests that cross origins (domain A -> B)
  - This is mainly applicable to XmlHttpRequests (AJAX)
  - But some resources have this as well, e.g. fonts & JS

- So when we’re loading resources, the browser will examine the response’ CORS headers to see if your origin (domain A) has been permitted to load stuff from this domain (domain B)

So How Does HTML Rendering & JS Execution Work?

- **Hah, nope, sorry,** that question is <u>too ambitious</u> even for this obnoxious over-the-top joke slide deck.

- Rendering a webpage in 2024 correctly and securely is the hardest problem that mankind has invented.
  - The moon? *Please, we did that with a TI-84.*

- Problem is so hard, Microsoft gave up on it & started using Chromium for Edge

So That's It

You now know what happens.

There's a lot more that could be discussed which wasn't relevant to the question, like cookies, POST, streaming, and more.

I HOPE THIS TALK HAS SHOWN YOU THE INCREDIBLE COMPLEXITY OF THE WORLD WE HAVE WROUGHT

IT IS YOUR JOB TO KEEP THE HOUSE OF CARDS STANDING FOR THE NEXT 40+ YEARS

please enjoy your career in technology!

Scan the QR code for a copy of this deck