

Ask or Search Quora

Ask Question

Read

Answer

Notifications

Subbu

Uniform Resource Locators (URLs) How Things Work Web Browsers +3

What are the series of steps that happen when an URL is requested from the address field of a browser?

Detailed steps on how the flow happens among Routers, DNS lookup servers, Web Servers etc

Answer

Request

Follow 192 Comments 1+ Share 2 Downvote

23 Answers



E.O. Stinson, Cloud control, bioinformatics, casual cyclist, hobbyist cook.
Bullet points.

Written May 3, 2010 · Upvoted by William Emmanuel Yu, computer networks teacher

This is a question whose answer could grow into an entire course on networking, so here's a version that only details some of the cases. There could probably be followup questions.

1. The browser extracts the domain name from the URL.
2. The browser queries DNS for the IP address of the URL. Generally, the browser will have cached domains previously visited, and the operating system will have cached queries from any number of applications. If neither the browser nor the OS have a cached copy of the IP address, then a request is sent off to the system's configured DNS server. The client machine knows the IP address for the DNS server, so no lookup is necessary.
3. The request sent to the DNS server is almost always smaller than the maximum packet size, and is thus sent off as a single packet. In addition to the content of the request, the packet includes the IP address it is destined for in its header. Except in the simplest of cases (network hubs), as the packet reaches each piece of network equipment between the client and server, that equipment uses a routing table to figure out what node it is connected to that is most likely to be part of the fastest route to the destination. The process of determining which path is the best choice differs between equipment and can be very complicated.
4. The is either lost (in which case the request fails or is reiterated), or makes it to its destination, the DNS server.
5. If that DNS server has the address for that domain, it will return it. Otherwise, it will forward the query along to DNS server it is configured to defer to. This happens recursively until the request is fulfilled or it reaches an authoritative name server and can go no further. (If the authoritative name server doesn't recognize the domain, the response indicates failure and the browser generally gives an error like "Can't find the server at www.lkliejafadh.com ".) The response makes its way back to the client machine much like the request traveled to the DNS server.
6. Assuming the DNS request is successful, the client machine now has an IP address that uniquely identifies a machine on the Internet. The web browser then assembles an HTTP request, which consists of a header and optional content. The header includes things like the specific path being requested from the web server, the HTTP version, any relevant browser cookies, etc. In

Upvote 264

Downvote Comments 5

There's more on Quora...

Pick new people and topics to follow and see the best answers on Quora.

Update Your Interests

Related Questions

What actually happens behind while we enter any URL at the address bar?

What are the troubleshooting steps that need to be taken when your browser displays the www.google.com webpage when you have entered www.yahoo...

Describe what happens in a browser once you hit enter after writing a URL in the address bar?

What is the life of a request(POST/GET) originating from a browser until it reaches its target server?

When you type a url (of a website say) and view something from it, how does that data transfer happen in a fraction of seconds?

What happens when you type <http://www.google.com/> in a web browser address bar and press enter?

What happens when you type Google in a web browser address bar and press enter?

What happens from the time we type the url to the time the page is routed?

When did browsers start letting you type in an url without the <http://> at the beginning?

What happens from the point you type in URL in browser to the point the page is displayed?

More Related Questions

Question Stats

192 Followers

157,937 Views

Last Asked Feb 18

10 Merged Questions

Edits

Ask or Search Quora

Ask Question

Read

Answer

Notifications

 Subbu

7. This HTTP request is sent off to the web server host as some number of packets, each of which is routed in the same way as the earlier DNS query. (The packets have sequence numbers that allow them to be reassembled in order even if they take different paths.) Once the request arrives at the webserver, it generates a response (this may be a static page, served as-is, or a more dynamic response, generated in any number of ways.) The web server software sends the generated page back to the client.
8. Assuming the response HTML and not an image or data file, then the browser parses the HTML to render the page. Part of this parsing and rendering process may be the discovery that the web page includes images or other embedded content that is not part of the HTML document. The browser will then send off further requests (either to the original web server or different ones, as appropriate) to fetch the embedded content, which will then be rendered into the document as well.

See also:

- <http://en.wikipedia.org/wiki/Dom...>
- <http://en.wikipedia.org/wiki/Rou...>
- <http://en.wikipedia.org/wiki/Web...>
- <http://en.wikipedia.org/wiki/HTML>

126.3k Views · View Upvotes

Segment - Official Site

Integrations & analytics made easy. Send data to 100+ apps. Free trial!

Free Trial at segment.com

Sumit Jha, Software developer, Science Lover

Written Apr 27, 2014

Originally Answered: What happens when you type "google.com" into your browser's address bar?

As far as i know....

When you enter google.com in the address bar of the browser then the following series of things happens

1. the browser need to know the numerical IP address so it first looks into its browser cache followed by OS cache, router cache, ISP DNS cache then a recursive search into ISP's DNS server begins with through the TLD nameserver until it finds the required ip address.

there is a concept of load balancer which also comes into play . it is just a piece of hardware that listens on a particular IP address and forwards the requests to other servers. Major sites will typically use expensive high-performance load balancers

2. after obtaining the IP the browser sends a HTTP request to the web server

3. the google server then responds with a permanent redirect (301) . it tells the browser to go "http://www.google.com/" instead of "http://google.com/"

Upvote 264

Downvote Comments 5

[Ask Question](#)[Read](#)[Answer](#)[Notifications](#)[Subbu](#)

downloading it as a file.

6.The browser begins rendering the HTML and sends the request for object embedded in HTML as many sites deliver their CSS,Images/Sprite files and scripts file from a content delivery network (**CDN**). the browser will again send the GET request for each of the embedded URL which again goes by the same procedure of look up and other above mention steps.

7. After this the browser may send further AJAX request to communicate with the web server even after the page is rendered.

so this is the bigger picture of how this works. there are many low level details which i left out intentionally (because i don't know about them :p)

50k Views · View Upvotes

[Upvote 71](#)[Downvote](#) [Comments 3+](#)

Ankush Chauhan, privacy is a myth about internet.

Written Oct 16, 2015



It's not as simple as it seems:

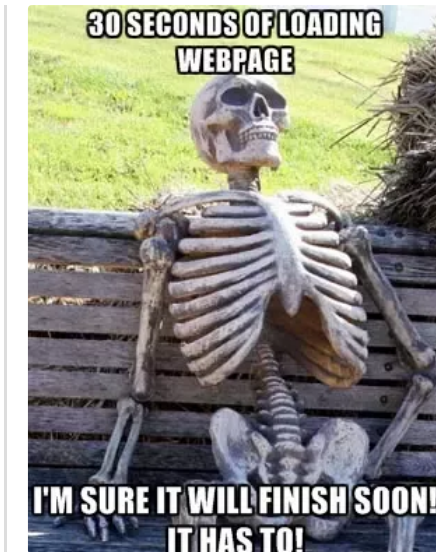
1. You **type an URL** into address bar in your preferred browser.
2. The browser **parses the URL** to find the protocol, host, port, and path.
3. It **forms a HTTP request** (that was most likely the protocol)
4. To reach the host, it first needs to **translate** the human readable host **into an IP number**, and it does this by doing a DNS lookup on the host
5. Then a **socket needs to be opened** from the user's computer to that IP number, on the port specified (most often port 80)
6. When a connection is open, the **HTTP request is sent** to the host
7. The host **forwards the request** to the server software (most often Apache) configured to listen on the specified port
8. The **server inspects the request** (most often only the path), and **launches the server plugin needed** to handle the request (corresponding to the server language you use, PHP, Java, .NET, Python?)

[Upvote 264](#)[Downvote](#) [Comments 5](#)

[Ask Question](#)[Read](#)[Answer](#)[Notifications](#)[Subbu](#)

11. Data from the database, together with other information the plugin decides to add, is **combined into a long string** of text (probably HTML).
12. The plugin **combines** that data with some meta data (in the form of HTTP headers), and **sends the HTTP response** back to the browser.
13. The browser receives the response, and **parses the HTML** (which with 95% probability is broken) in the response
14. A **DOM tree is built** out of the broken HTML
15. **New requests are made** to the server for each new resource that is found in the HTML source (typically images, style sheets, and JavaScript files). Go back to step 3 and repeat for each resource.
16. **Stylesheets are parsed**, and the rendering information in each gets attached to the matching node in the DOM tree
17. **Javascript is parsed and executed**, and DOM nodes are moved and style information is updated accordingly
18. The browser **renders the page** on the screen according to the DOM tree and the style information for each node
19. **You see** the page on the screen
20. You get annoyed the whole process was too slow.

I, too, get annoyed when the above steps take longer than one tenth of a second. But now at least I have some documentation to look at, while waiting the remaining fractions of a second before the page renders.



17.7k Views · View Upvotes · Answer requested by Basavaraj Hg

[Upvote](#) 37[Downvote](#) [Comment](#) 1

Let us build your dream mobile app today.

Hire the premier Silicon Valley Based Mobile App Development Company to see your app take flight.

[Get Quote at loka.com](#)

Bageshwar Pratap Narain Software Engineer

[Upvote](#) 264[Downvote](#) [Comments](#) 5

[Ask Question](#)[Read](#)[Answer](#)[Notifications](#)[Subbu](#)

Augmenting the previous answer by Shanmugasundaram-Muthuswamy

If we go further down the "technical" road. HTTP is designed on top of TCP IP, so once the ip address of [Google](#) is resolved ,a tcp hand shake will take place.
since google defaults to HTTPS, a SSL handshake will also take place, and the public key will be exchanged, which will be used to encrypt /decrypt further requests.

Technically making an HTTP request can be explained differently in all the different layers of the network.

10.8k Views · View Upvotes

[Upvote 6](#)[Downvote](#) [Comment 1](#)

Top Stories from Your Feed

 Bruce R. Miller wrote this · 2015


How can I relax for my full day interview with Google?



Bruce R. Miller, works at Google
Written Jul 28, 2015 · Upvoted by
Abhishek Kumar, Software Engineer at Google, Andre Tacuyan, works at Google, and Mayank S

At the beginning of each interview, I tell the candidate, "Pretend you've already been hired and I am the engineer in the office next to yours who needs help with a problem." The most difficult pa...

[Read In Feed](#)

 Drew Eckhardt upvoted this · Sat

As a software engineer, do the master's and PhD degrees make me a better engineer or just better at focusing on my career skills?



John L. Miller, 25 years at:
Microsoft, CMU, Amazon, Google, Oracle, JPRC, etc.: PhD in C.S.
Written Sat · Upvoted by Drew Eckhardt

Many people getting degrees view them as a way to prove a certain skill level. Novice software engineers hoping to work at the big software companies are no different.

[Read In Feed](#)

Answer written · 2016

What are the negatives in working in a too-good-to-be-true office like Google?



Sadia Hdydi, IT professional and single mother
Written Jun 27, 2016 · Upvoted by James Beveridge, worked at Google and Kevin X Chang, ex-Google, ex-YouTube

I was offered a job at a company (not Google) with similar perks: free food on site, a margarita machine that was always on, campus gym, etc. I was in my 30s and a single mother. Those "perks" were...

[Read In Feed](#)[Upvote 264](#)[Downvote](#) [Comments 5](#)

what happens when you type in a URL in browser [closed]

Can somebody tell me what all happens behind the scenes from the time I type in a URL in the browser to the time when I get to see the page on the browser? A detailed account of the process would be of great help.

browser

edited Feb 3 '16 at 1:26

asked Jan 19 '10 at 9:46



Eye

3,513 2 22 49



Aadith

3,391 11 41 72

closed as off topic by [skaffman](#), [Dominic Rodger](#), [Jarrett Meyer](#), [Graviton](#), [SilentGhost](#) Jan 19 '10 at 15:30

Questions on Stack Overflow are expected to relate to programming within the scope [defined by the community](#). Consider editing the question or leaving comments for improvement if you believe the question can be reworded to fit within the scope. Read more about [reopening questions](#) here.

If this question can be reworded to fit the rules in the [help center](#), please [edit the question](#).

2 Though this may be programming related (eventually) - the level of detail to which this could be answered would (and has) filled volumes. Please restate as a programming query. – [KevinDTimm](#) Jan 19 '10 at 9:48

15 Get O'Reilly's *DNS and Bind* book. It's only 624 pages. – [Wim Hollebrandse](#) Jan 19 '10 at 9:57

6 [edusagar.com/articles/view/70/...](#) this is the best possible answer! – [Shivendra](#) Aug 24 '14 at 5:25

1 For posterity's sake, here is a detailed version of how the internet works - [goo.gl/eEHmpZ](#). – [Ashwin Krishnamurthy](#) Oct 7 '14 at 18:49

4 There's now a collaborative effort to answer this in as much detail as possible: [github.com/alex/what-happens-when/blob/master/README.rst](#) – [Piskvor](#) Mar 12 '15 at 8:34

3 Answers

In an extremely rough and simplified sketch, assuming the simplest possible HTTP request, no proxies, IPv4 and no problems in any step:

1. browser checks cache; if requested object is in cache and is fresh, skip to #9
2. browser asks OS for server's IP address
3. OS makes a DNS lookup and replies the IP address to the browser
4. browser opens a TCP connection to server (this step is much more complex with HTTPS)
5. browser sends the HTTP request through TCP connection
6. browser receives HTTP response and may close the TCP connection, or reuse it for another request
7. browser checks if the response is a redirect or a conditional response (3xx result status codes), authorization request (401), error (4xx and 5xx), etc.; these are handled differently from normal responses (2xx)
8. if cacheable, response is stored in cache
9. browser decodes response (e.g. if it's gzipped)
10. browser determines what to do with response (e.g. is it a HTML page, is it an image, is it a sound clip?)
11. browser renders response, or offers a download dialog for unrecognized types

Again, discussion of each of these points have filled countless pages; take this only as a short summary. Also, there are many other things happening in parallel to this (processing typed-in address, speculative prefetching, adding page to browser history, displaying progress to user, notifying plugins and extensions, rendering the page while it's downloading, pipelining, connection tracking for keep-alive, checking for malicious content etc.) - and the whole operation gets an order of magnitude more complex with HTTPS (certificates and ciphers and pinning, oh my!).

edited Dec 14 '15 at 9:43

answered Jan 19 '10 at 10:01



Piskvor

64.2k 40 140 189

1 Wow! Amazingly explained. – [gaurav414u](#) Nov 27 '15 at 13:41

First the computer looks up the destination host. If it exists in local DNS cache, it uses that information. Otherwise, DNS querying is performed until the IP address is found.

Then, your browser opens a TCP connection to the destination host and sends the request according to HTTP 1.1 (or might use HTTP 1.0, but normal browsers don't do it any more).

The server looks up the required resource (if it exists) and responds using HTTP protocol, sends the data to the client (=your browser)

The browser then uses HTML parser to re-create document structure which is later presented to you on screen. If it finds references to external resources, such as pictures, css files, javascript files, these are delivered the same way as the HTML document itself.

answered Jan 19 '10 at 9:57



[naivists](#)

21.6k 4 39 72

Look up the specification of HTTP. Or to get started, try
<http://www.jmarshall.com/easy/http/>

answered Jan 19 '10 at 9:54



[John](#)

4,927 2 22 43

This repository | Search

Pull requests Issues Gist



alex / what-happens-when

Watch 638

Star 13,796

Fork 986

Code

Issues 64

Pull requests 28

Projects 0

Wiki

Pulse

Graphs

An attempt to answer the age old interview question "What happens when you type google.com into your browser and press enter?"

278 commits

1 branch

0 releases

61 contributors

Branch: master

New pull request

Create new file

Upload files

Find file

Clone or download

marumari Merge branch 'master' of github.com:alex/what-happens-when ...

Latest commit bde28a2 on Dec 19, 2016

.travis.yml Add more detail to page rendering and add links to specs.

2 years ago

README.rst Proper abbreviation, closes #223

3 months ago

README.rst

What happens when...

This repository is an attempt to answer the age old interview question "What happens when you type google.com into your browser's address box and press enter?"

Except instead of the usual story, we're going to try to answer this question in as much detail as possible. No skipping out on anything.

This is a collaborative process, so dig in and try to help out! There's tons of details missing, just waiting for you to add them! So send us a pull request, please!

This is all licensed under the terms of the [Creative Commons Zero](#) license.

Read this in [简体中文](#) (simplified Chinese). NOTE: this has not been reviewed by the alex/what-happens-when maintainers.

Table of Contents

- [The "g" key is pressed](#)
- [The "enter" key bottoms out](#)
- [Interrupt fires \[NOT for USB keyboards\]](#)
- [\(On Windows\) A WM_KEYDOWN message is sent to the app](#)
- [\(On OS X\) A KeyDown NSEvent is sent to the app](#)
- [\(On GNU/Linux\) the Xorg server listens for keycodes](#)
- [Parse URL](#)
- [Is it a URL or a search term?](#)
- [Convert non-ASCII Unicode characters in hostname](#)
- [Check HSTS list](#)
- [DNS lookup](#)
- [ARP process](#)
- [Opening of a socket](#)

- [TLS handshake](#)
- [HTTP protocol](#)
- [HTTP Server Request Handle](#)
- [Behind the scenes of the Browser](#)
- [Browser](#)
- [HTML parsing](#)
- [CSS interpretation](#)
- [Page Rendering](#)
- [GPU Rendering](#)
- [Window Server](#)
- [Post-rendering and user-induced execution](#)

The "g" key is pressed

The following sections explain all about the physical keyboard and the OS interrupts. But, a whole lot happens after that which isn't explained. When you just press "g" the browser receives the event and the entire auto-complete machinery kicks into high gear. Depending on your browser's algorithm and if you are in private/incognito mode or not various suggestions will be presented to you in the dropdown below the URL bar. Most of these algorithms prioritize results based on search history and bookmarks. You are going to type "google.com" so none of it matters, but a lot of code will run before you get there and the suggestions will be refined with each key press. It may even suggest "google.com" before you type it.

The "enter" key bottoms out

To pick a zero point, let's choose the Enter key on the keyboard hitting the bottom of its range. At this point, an electrical circuit specific to the enter key is closed (either directly or capacitively). This allows a small amount of current to flow into the logic circuitry of the keyboard, which scans the state of each key switch, debounces the electrical noise of the rapid intermittent closure of the switch, and converts it to a keycode integer, in this case 13. The keyboard controller then encodes the keycode for transport to the computer. This is now almost universally over a Universal Serial Bus (USB) or Bluetooth connection, but historically has been over PS/2 or ADB connections.

In the case of the USB keyboard:

- The USB circuitry of the keyboard is powered by the 5V supply provided over pin 1 from the computer's USB host controller.
- The keycode generated is stored by internal keyboard circuitry memory in a register called "endpoint".
- The host USB controller polls that "endpoint" every ~10ms (minimum value declared by the keyboard), so it gets the keycode value stored on it.
- This value goes to the USB SIE (Serial Interface Engine) to be converted in one or more USB packets that follow the low level USB protocol.
- Those packets are sent by a differential electrical signal over D+ and D- pins (the middle 2) at a maximum speed of 1.5 Mb/s, as an HID (Human Interface Device) device is always declared to be a "low speed device" (USB 2.0 compliance).
- This serial signal is then decoded at the computer's host USB controller, and interpreted by the computer's Human Interface Device (HID) universal keyboard device driver. The value of the key is then passed into the operating system's hardware abstraction layer.

In the case of Virtual Keyboard (as in touch screen devices):

- When the user puts their finger on a modern capacitive touch screen, a tiny amount of current gets transferred to the finger. This completes the circuit through the electrostatic field of the conductive layer and creates a voltage drop at that point on the screen. The screen controller then raises an interrupt reporting the coordinate of the key press.
- Then the mobile OS notifies the current focused application of a press event in one of its GUI elements (which now is the virtual keyboard application buttons).

- The virtual keyboard can now raise a software interrupt for sending a 'key pressed' message back to the OS.
- This interrupt notifies the current focused application of a 'key pressed' event.

Interrupt fires [NOT for USB keyboards]

The keyboard sends signals on its interrupt request line (IRQ), which is mapped to an `interrupt vector` (integer) by the interrupt controller. The CPU uses the `Interrupt Descriptor Table (IDT)` to map the interrupt vectors to functions (`interrupt handlers`) which are supplied by the kernel. When an interrupt arrives, the CPU indexes the IDT with the interrupt vector and runs the appropriate handler. Thus, the kernel is entered.

(On Windows) A **WM_KEYDOWN** message is sent to the app

The HID transport passes the key down event to the `KBDHID.sys` driver which converts the HID usage into a scancode. In this case the scan code is `VK_RETURN (0x0D)`. The `KBDHID.sys` driver interfaces with the `KBDCLASS.sys` (keyboard class driver). This driver is responsible for handling all keyboard and keypad input in a secure manner. It then calls into `Win32K.sys` (after potentially passing the message through 3rd party keyboard filters that are installed). This all happens in kernel mode.

`Win32K.sys` figures out what window is the active window through the `GetForegroundWindow()` API. This API provides the window handle of the browser's address box. The main Windows "message pump" then calls `SendMessage(hWnd, WM_KEYDOWN, VK_RETURN, lParam)`. `lParam` is a bitmask that indicates further information about the keypress: repeat count (0 in this case), the actual scan code (can be OEM dependent, but generally wouldn't be for `VK_RETURN`), whether extended keys (e.g. alt, shift, ctrl) were also pressed (they weren't), and some other state.

The Windows `SendMessage` API is a straightforward function that adds the message to a queue for the particular window handle (`hWnd`). Later, the main message processing function (called a `WindowProc`) assigned to the `hWnd` is called in order to process each message in the queue.

The window (`hWnd`) that is active is actually an edit control and the `WindowProc` in this case has a message handler for `WM_KEYDOWN` messages. This code looks within the 3rd parameter that was passed to `SendMessage (wParam)` and, because it is `VK_RETURN` knows the user has hit the ENTER key.

(On OS X) A **KeyDown** NSEvent is sent to the app

The interrupt signal triggers an interrupt event in the I/O Kit keyboard driver. The driver translates the signal into a key code which is passed to the OS X `WindowServer` process. Resultantly, the `WindowServer` dispatches an event to any appropriate (e.g. active or listening) applications through their Mach port where it is placed into an event queue. Events can then be read from this queue by threads with sufficient privileges calling the `mach_ipc_dispatch` function. This most commonly occurs through, and is handled by, an `NSApplication` main event loop, via an `NSEvent` of `NSEventType KeyDown`.

(On GNU/Linux) the Xorg server listens for keycodes

When a graphical `X server` is used, `X` will use the generic event driver `evdev` to acquire the keypress. A re-mapping of keycodes to scan codes is made with `X server` specific keymaps and rules. When the scan code mapping of the key pressed is complete, the `X server` sends the character to the `window manager` (`DWM`, `metacity`, `i3`, etc), so the `window manager` in turn sends the character to the focused window. The graphical API of the window that receives the character prints the appropriate font symbol in the appropriate focused field.

Parse URL

- The browser now has the following information contained in the URL (Uniform Resource Locator):
 - **Protocol** `"http"`
Use 'Hyper Text Transfer Protocol'
 - **Resource** `"/"`
Retrieve main (index) page

Is it a URL or a search term?

When no protocol or valid domain name is given the browser proceeds to feed the text given in the address box to the browser's default web search engine. In many cases the url has a special piece of text appended to it to tell the search engine that it came from a particular browser's url bar.

Convert non-ASCII Unicode characters in hostname

- The browser checks the hostname for characters that are not in `a-z` , `A-Z` , `0-9` , `-` , or `.` .
- Since the hostname is `google.com` there won't be any, but if there were the browser would apply [Punycode](#) encoding to the hostname portion of the URL.

Check HSTS list

- The browser checks its "preloaded HSTS (HTTP Strict Transport Security)" list. This is a list of websites that have requested to be contacted via HTTPS only.
- If the website is in the list, the browser sends its request via HTTPS instead of HTTP. Otherwise, the initial request is sent via HTTP. (Note that a website can still use the HSTS policy *without* being in the HSTS list. The first HTTP request to the website by a user will receive a response requesting that the user only send HTTPS requests. However, this single HTTP request could potentially leave the user vulnerable to a [downgrade attack](#), which is why the HSTS list is included in modern web browsers.)

DNS lookup

- Browser checks if the domain is in its cache. (to see the DNS Cache in Chrome, go to `chrome://net-internals/#dns`).
- If not found, the browser calls `gethostbyname` library function (varies by OS) to do the lookup.
- `gethostbyname` checks if the hostname can be resolved by reference in the local `hosts` file (whose location [varies by OS](#)) before trying to resolve the hostname through DNS.
- If `gethostbyname` does not have it cached nor can find it in the `hosts` file then it makes a request to the DNS server configured in the network stack. This is typically the local router or the ISP's caching DNS server.
- If the DNS server is on the same subnet the network library follows the `ARP` process below for the DNS server.
- If the DNS server is on a different subnet, the network library follows the `ARP` process below for the default gateway IP.

ARP process

In order to send an ARP (Address Resolution Protocol) broadcast the network stack library needs the target IP address to look up. It also needs to know the MAC address of the interface it will use to send out the ARP broadcast.

The ARP cache is first checked for an ARP entry for our target IP. If it is in the cache, the library function returns the result: Target IP = MAC.

If the entry is not in the ARP cache:

- The route table is looked up, to see if the Target IP address is on any of the subnets on the local route table. If it is, the library uses the interface associated with that subnet. If it is not, the library uses the interface that has the subnet of our default gateway.
- The MAC address of the selected network interface is looked up.
- The network library sends a Layer 2 (data link layer of the [OSI model](#)) ARP request:

ARP Request :

```
Sender MAC: interface:mac:address:here
Sender IP: interface.ip.goes.here
Target MAC: FF:FF:FF:FF:FF:FF (Broadcast)
Target IP: target.ip.goes.here
```

Depending on what type of hardware is between the computer and the router:

Directly connected:

- If the computer is directly connected to the router the router responds with an ARP Reply (see below)

Hub:

- If the computer is connected to a hub, the hub will broadcast the ARP request out all other ports. If the router is connected on the same "wire", it will respond with an ARP Reply (see below).

Switch:

- If the computer is connected to a switch, the switch will check its local CAM/MAC table to see which port has the MAC address we are looking for. If the switch has no entry for the MAC address it will rebroadcast the ARP request to all other ports.
- If the switch has an entry in the MAC/CAM table it will send the ARP request to the port that has the MAC address we are looking for.
- If the router is on the same "wire", it will respond with an ARP Reply (see below)

ARP Reply :

```
Sender MAC: target:mac:address:here
Sender IP: target.ip.goes.here
Target MAC: interface:mac:address:here
Target IP: interface.ip.goes.here
```

Now that the network library has the IP address of either our DNS server or the default gateway it can resume its DNS process:

- Port 53 is opened to send a UDP request to DNS server (if the response size is too large, TCP will be used instead).
- If the local/ISP DNS server does not have it, then a recursive search is requested and that flows up the list of DNS servers until the SOA is reached, and if found an answer is returned.

Opening of a socket

Once the browser receives the IP address of the destination server, it takes that and the given port number from the URL (the HTTP protocol defaults to port 80, and HTTPS to port 443), and makes a call to the system library function named `socket` and requests a TCP socket stream - `AF_INET/AF_INET6` and `SOCK_STREAM` .

- This request is first passed to the Transport Layer where a TCP segment is crafted. The destination port is added to the header, and a source port is chosen from within the kernel's dynamic port range (`ip_local_port_range` in Linux).
- This segment is sent to the Network Layer, which wraps an additional IP header. The IP address of the destination server as well as that of the current machine is inserted to form a packet.
- The packet next arrives at the Link Layer. A frame header is added that includes the MAC address of the machine's NIC as well as the MAC address of the gateway (local router). As before, if the kernel does not know the MAC address of the gateway, it must broadcast an ARP query to find it.

At this point the packet is ready to be transmitted through either:

- [Ethernet](#)
- [WiFi](#)
- [Cellular data network](#)

For most home or small business Internet connections the packet will pass from your computer, possibly through a local network, and then through a modem (MOdulator/DEModulator) which converts digital 1's and 0's into an analog signal suitable for transmission over telephone, cable, or wireless telephony connections. On the other end of the

connection is another modem which converts the analog signal back into digital data to be processed by the next [network node](#) where the from and to addresses would be analyzed further.

Most larger businesses and some newer residential connections will have fiber or direct Ethernet connections in which case the data remains digital and is passed directly to the next [network node](#) for processing.

Eventually, the packet will reach the router managing the local subnet. From there, it will continue to travel to the autonomous system's (AS) border routers, other ASes, and finally to the destination server. Each router along the way extracts the destination address from the IP header and routes it to the appropriate next hop. The time to live (TTL) field in the IP header is decremented by one for each router that passes. The packet will be dropped if the TTL field reaches zero or if the current router has no space in its queue (perhaps due to network congestion).

This send and receive happens multiple times following the TCP connection flow:

- Client chooses an initial sequence number (ISN) and sends the packet to the server with the SYN bit set to indicate it is setting the ISN
- *Server receives SYN and if it's in an agreeable mood:*
 - Server chooses its own initial sequence number
 - Server sets SYN to indicate it is choosing its ISN
 - Server copies the (client ISN +1) to its ACK field and adds the ACK flag to indicate it is acknowledging receipt of the first packet
- *Client acknowledges the connection by sending a packet:*
 - Increases its own sequence number
 - Increases the receiver acknowledgment number
 - Sets ACK field
- *Data is transferred as follows:*
 - As one side sends N data bytes, it increases its SEQ by that number
 - When the other side acknowledges receipt of that packet (or a string of packets), it sends an ACK packet with the ACK value equal to the last received sequence from the other
- *To close the connection:*
 - The closer sends a FIN packet
 - The other sides ACKs the FIN packet and sends its own FIN
 - The closer acknowledges the other side's FIN with an ACK

TLS handshake

- The client computer sends a `ClientHello` message to the server with its Transport Layer Security (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 signed by a CA (Certificate Authority). 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 against its list of trusted CAs. If trust can be established based on the CA, the client generates a string of pseudo-random bytes and encrypts this with the server's public key. These random bytes can be used to determine the symmetric key.
- The server decrypts the random bytes using its private key and uses these bytes to generate its own copy of the symmetric master 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.
- From now on the TLS session transmits the application (HTTP) data encrypted with the agreed symmetric key.

HTTP protocol

If the web browser used was written by Google, instead of sending an HTTP request to retrieve the page, it will send a request to try and negotiate with the server an "upgrade" from HTTP to the SPDY protocol.

If the client is using the HTTP protocol and does not support SPDY, it sends a request to the server of the form:

```
GET / HTTP/1.1
Host: google.com
Connection: close
[other headers]
```

where [other headers] refers to a series of colon-separated key-value pairs formatted as per the HTTP specification and separated by single new lines. (This assumes the web browser being used doesn't have any bugs violating the HTTP spec. This also assumes that the web browser is using HTTP/1.1, otherwise it may not include the Host header in the request and the version specified in the GET request will either be HTTP/1.0 or HTTP/0.9.)

HTTP/1.1 defines the "close" connection option for the sender to signal that the connection will be closed after completion of the response. For example,

```
Connection: close
```

HTTP/1.1 applications that do not support persistent connections MUST include the "close" connection option in every message.

After sending the request and headers, the web browser sends a single blank newline to the server indicating that the content of the request is done.

The server responds with a response code denoting the status of the request and responds with a response of the form:

```
200 OK
[response headers]
```

Followed by a single newline, and then sends a payload of the HTML content of `www.google.com`. The server may then either close the connection, or if headers sent by the client requested it, keep the connection open to be reused for further requests.

If the HTTP headers sent by the web browser included sufficient information for the web server to determine if the version of the file cached by the web browser has been unmodified since the last retrieval (ie. if the web browser included an ETag header), it may instead respond with a request of the form:

```
304 Not Modified
[response headers]
```

and no payload, and the web browser instead retrieves the HTML from its cache.

After parsing the HTML, the web browser (and server) repeats this process for every resource (image, CSS, favicon.ico, etc) referenced by the HTML page, except instead of GET / HTTP/1.1 the request will be GET /\$(URL relative to www.google.com) HTTP/1.1.

If the HTML referenced a resource on a different domain than `www.google.com`, the web browser goes back to the steps involved in resolving the other domain, and follows all steps up to this point for that domain. The Host header in the request will be set to the appropriate server name instead of `google.com`.

HTTP Server Request Handle

The HTTPD (HTTP Daemon) server is the one handling the requests/responses on the server side. The most common HTTPD servers are Apache or nginx for Linux and IIS for Windows.

- The HTTPD (HTTP Daemon) receives the request.
- *The server breaks down the request to the following parameters:*
 - HTTP Request Method (either GET , HEAD , POST , PUT , DELETE , CONNECT , OPTIONS , or TRACE). In the case of a URL entered directly into the address bar, this will be GET .
 - Domain, in this case - google.com.
 - Requested path/page, in this case - / (as no specific path/page was requested, / is the default path).
- The server verifies that there is a Virtual Host configured on the server that corresponds with google.com.
- The server verifies that google.com can accept GET requests.
- The server verifies that the client is allowed to use this method (by IP, authentication, etc.).
- If the server has a rewrite module installed (like mod_rewrite for Apache or URL Rewrite for IIS), it tries to match the request against one of the configured rules. If a matching rule is found, the server uses that rule to rewrite the request.
- The server goes to pull the content that corresponds with the request, in our case it will fall back to the index file, as "/" is the main file (some cases can override this, but this is the most common method).
- The server parses the file according to the handler. If Google is running on PHP, the server uses PHP to interpret the index file, and streams the output to the client.

Behind the scenes of the Browser

Once the server supplies the resources (HTML, CSS, JS, images, etc.) to the browser it undergoes the below process:

- Parsing - HTML, CSS, JS
- Rendering - Construct DOM Tree → Render Tree → Layout of Render Tree → Painting the render tree

Browser

The browser's functionality is to present the web resource you choose, by requesting it from the server and displaying it in the browser window. The resource is usually an HTML document, but may also be a PDF, image, or some other type of content. The location of the resource is specified by the user using a URI (Uniform Resource Identifier).

The way the browser interprets and displays HTML files is specified in the HTML and CSS specifications. These specifications are maintained by the W3C (World Wide Web Consortium) organization, which is the standards organization for the web.

Browser user interfaces have a lot in common with each other. Among the common user interface elements are:

- An address bar for inserting a URI
- Back and forward buttons
- Bookmarking options
- Refresh and stop buttons for refreshing or stopping the loading of current documents
- Home button that takes you to your home page

Browser High Level Structure

The components of the browsers are:

- User interface: The user interface includes the address bar, back/forward button, bookmarking menu, etc. Every part of the browser display except the window where you see the requested page.
- Browser engine: The browser engine marshals actions between the UI and the rendering engine.
- Rendering engine: The rendering engine is responsible for displaying requested content. For example if the requested content is HTML, the rendering engine parses HTML and CSS, and displays the parsed content on the screen.

- **Networking:** The networking handles network calls such as HTTP requests, using different implementations for different platforms behind a platform-independent interface.
- **UI backend:** The UI backend is used for drawing basic widgets like combo boxes and windows. This backend exposes a generic interface that is not platform specific. Underneath it uses operating system user interface methods.
- **JavaScript engine:** The JavaScript engine is used to parse and execute JavaScript code.
- **Data storage:** The data storage is a persistence layer. The browser may need to save all sorts of data locally, such as cookies. Browsers also support storage mechanisms such as localStorage, IndexedDB, WebSQL and FileSystem.

HTML parsing

The rendering engine starts getting the contents of the requested document from the networking layer. This will usually be done in 8kB chunks.

The primary job of HTML parser to parse the HTML markup into a parse tree.

The output tree (the "parse tree") is a tree of DOM element and attribute nodes. DOM is short for Document Object Model. It is the object presentation of the HTML document and the interface of HTML elements to the outside world like JavaScript. The root of the tree is the "Document" object. Prior of any manipulation via scripting, the DOM has an almost one-to-one relation to the markup.

The parsing algorithm

HTML cannot be parsed using the regular top-down or bottom-up parsers.

The reasons are:

- The forgiving nature of the language.
- The fact that browsers have traditional error tolerance to support well known cases of invalid HTML.
- The parsing process is reentrant. For other languages, the source doesn't change during parsing, but in HTML, dynamic code (such as script elements containing document.write() calls) can add extra tokens, so the parsing process actually modifies the input.

Unable to use the regular parsing techniques, the browser utilizes a custom parser for parsing HTML. The parsing algorithm is described in detail by the HTML5 specification.

The algorithm consists of two stages: tokenization and tree construction.

Actions when the parsing is finished

The browser begins fetching external resources linked to the page (CSS, images, JavaScript files, etc.).

At this stage the browser marks the document as interactive and starts parsing scripts that are in "deferred" mode: those that should be executed after the document is parsed. The document state is set to "complete" and a "load" event is fired.

Note there is never an "Invalid Syntax" error on an HTML page. Browsers fix any invalid content and go on.

CSS interpretation

- Parse CSS files, <style> tag contents, and style attribute values using ["CSS lexical and syntax grammar"](#)
- Each CSS file is parsed into a StyleSheet object, where each object contains CSS rules with selectors and objects corresponding CSS grammar.
- A CSS parser can be top-down or bottom-up when a specific parser generator is used.

Page Rendering

- Create a 'Frame Tree' or 'Render Tree' by traversing the DOM nodes, and calculating the CSS style values for each node.
- Calculate the preferred width of each node in the 'Frame Tree' bottom up by summing the preferred width of the child nodes and the node's horizontal margins, borders, and padding.
- Calculate the actual width of each node top-down by allocating each node's available width to its children.
- Calculate the height of each node bottom-up by applying text wrapping and summing the child node heights and the node's margins, borders, and padding.
- Calculate the coordinates of each node using the information calculated above.
- More complicated steps are taken when elements are floated, positioned absolutely or relatively, or other complex features are used. See <http://dev.w3.org/csswg/css2/> and <http://www.w3.org/Style/CSS/current-work> for more details.
- Create layers to describe which parts of the page can be animated as a group without being re-rasterized. Each frame/render object is assigned to a layer.
- Textures are allocated for each layer of the page.
- The frame/render objects for each layer are traversed and drawing commands are executed for their respective layer. This may be rasterized by the CPU or drawn on the GPU directly using D2D/SkiaGL.
- All of the above steps may reuse calculated values from the last time the webpage was rendered, so that incremental changes require less work.
- The page layers are sent to the compositing process where they are combined with layers for other visible content like the browser chrome, iframes and addon panels.
- Final layer positions are computed and the composite commands are issued via Direct3D/OpenGL. The GPU command buffer(s) are flushed to the GPU for asynchronous rendering and the frame is sent to the window server.

GPU Rendering

- During the rendering process the graphical computing layers can use general purpose CPU or the graphical processor GPU as well.
- When using GPU for graphical rendering computations the graphical software layers split the task into multiple pieces, so it can take advantage of GPU massive parallelism for float point calculations required for the rendering process.

Window Server

Post-rendering and user-induced execution

After rendering has completed, the browser executes JavaScript code as a result of some timing mechanism (such as a Google Doodle animation) or user interaction (typing a query into the search box and receiving suggestions). Plugins such as Flash or Java may execute as well, although not at this time on the Google homepage. Scripts can cause additional network requests to be performed, as well as modify the page or its layout, causing another round of page rendering and painting.



- [Home](#)
- [About](#)

[Igor Ostrovsky Blogging](#)

[On programming, technology, and random things of interest](#)

- [Algorithms](#)
- [Cool Stuff](#)
- [C#](#)
- [RoboZZle](#)
- [Concurrency](#)
- [Misc](#)

Feb
9

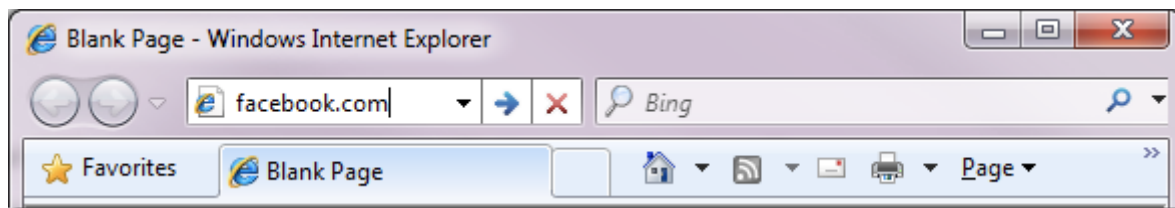
What really happens when you navigate to a URL

As a software developer, you certainly have a high-level picture of how web apps work and what kinds of technologies are involved: the browser, HTTP, HTML, web server, request handlers, and so on.

In this article, we will take a deeper look at the sequence of events that take place when you visit a URL.

1. You enter a URL into the browser

It all starts here:



2. The browser looks up the IP address for the domain name

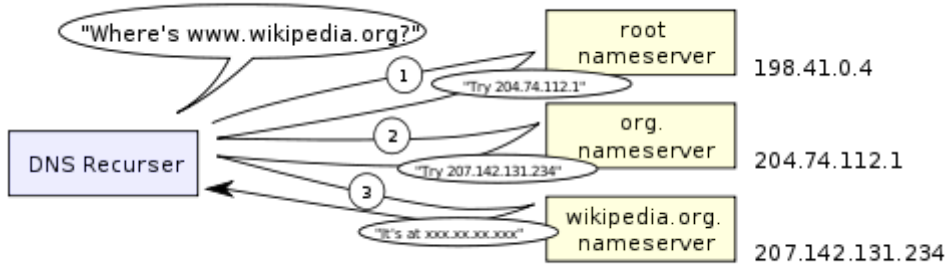


The first step in the navigation is to figure out the IP address for the visited domain. The DNS lookup proceeds as follows:

- **Browser cache** – The browser caches DNS records for some time. Interestingly, the OS does not tell the browser the time-to-live for each DNS record, and so the browser caches them for a fixed duration (varies between browsers, 2 – 30 minutes).
- **OS cache** – If the browser cache does not contain the desired record, the browser makes a system call (gethostbyname in Windows). The OS has its own cache.
- **Router cache** – The request continues on to your router, which typically has its own DNS cache.

- **ISP DNS cache** – The next place checked is the cache ISP's DNS server. With a cache, naturally.
- **Recursive search** – Your ISP's DNS server begins a recursive search, from the root nameserver, through the .com top-level nameserver, to Facebook's nameserver. Normally, the DNS server will have names of the .com nameservers in cache, and so a hit to the root nameserver will not be necessary.

Here is a diagram of what a recursive DNS search looks like:



One worrying thing about DNS is that the entire domain like wikipedia.org or facebook.com seems to map to a single IP address. Fortunately, there are ways of mitigating the bottleneck:

- **Round-robin DNS** is a solution where the DNS lookup returns multiple IP addresses, rather than just one. For example, facebook.com actually maps to four IP addresses.
- **Load-balancer** is the piece of hardware that listens on a particular IP address and forwards the requests to other servers. Major sites will typically use expensive high-performance load balancers.
- **Geographic DNS** improves scalability by mapping a domain name to different IP addresses, depending on the client's geographic location. This is great for hosting static content so that different servers don't have to update shared state.
- **Anycast** is a routing technique where a single IP address maps to multiple physical servers. Unfortunately, anycast does not fit well with TCP and is rarely used in that scenario.

Most of the DNS servers themselves use anycast to achieve high availability and low latency **of the DNS lookups**.

3. The browser sends a HTTP request to the web server



You can be pretty sure that Facebook's homepage will not be served from the browser cache because dynamic pages expire either very quickly or immediately (expiry date set to past).

So, the browser will send this request to the Facebook server:

```
GET http://facebook.com/ HTTP/1.1
Accept: application/x-ms-application, image/jpeg, application/xaml+xml, [...]
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64; [...])
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
Host: facebook.com
Cookie: datr=1265876274-[...]; locale=en_US; lsd=WW[...]; c_user=2101[...]
```

The GET request names the **URL** to fetch: “http://facebook.com/”. The browser identifies itself (**User-Agent** header), and states what types of responses it will accept (**Accept** and **Accept-Encoding** headers). The **Connection** header asks the server to keep the TCP connection open for further requests.

The request also contains the **cookies** that the browser has for this domain. As you probably already know, cookies are key-value pairs that track the state of a web site in between different page requests. And so the cookies store the name of the logged-in user, a secret number that was assigned to the user by the server, some of user’s settings, etc. The cookies will be stored in a text file on the client, and sent to the server with every request.

There is a variety of tools that let you view the raw HTTP requests and corresponding responses. My favorite tool for viewing the raw HTTP traffic is [fiddler](#), but there are many other tools (e.g., FireBug). These tools are a great help when optimizing a site.

In addition to GET requests, another type of requests that you may be familiar with is a POST request, typically used to submit forms. A GET request sends its parameters via the URL (e.g.: http://robozzle.com/puzzle.aspx?id=85). A POST request sends its parameters in the request body, just under the headers.

The trailing slash in the URL “http://facebook.com/” is important. In this case, the browser can safely add the slash. For URLs of the form http://example.com/folderOrFile, the browser cannot automatically add a slash, because it is not clear whether folderOrFile is a folder or a file. In such cases, the browser will visit the URL without the slash, and the server will respond with a redirect, resulting in an unnecessary roundtrip.

4. The facebook server responds with a permanent redirect



This is the response that the Facebook server sent back to the browser request:

```
HTTP/1.1 301 Moved Permanently
Cache-Control: private, no-store, no-cache, must-revalidate, post-check=0,
pre-check=0
Expires: Sat, 01 Jan 2000 00:00:00 GMT
```

```

Location: http://www.facebook.com/
P3P: CP="DSP LAW"
Pragma: no-cache
Set-Cookie: made_write_conn=deleted; expires=Thu, 12-Feb-2009 05:09:50 GMT;
           path=/; domain=.facebook.com; httponly
Content-Type: text/html; charset=utf-8
X-Cnection: close
Date: Fri, 12 Feb 2010 05:09:51 GMT
Content-Length: 0

```

The server responded with a 301 Moved Permanently response to tell the browser to go to “<http://www.facebook.com/>” instead of “<http://facebook.com/>”.

There are interesting reasons why the server insists on the redirect instead of immediately responding with the web page that the user wants to see.

One reason has to do with **search engine rankings**. See, if there are two URLs for the same page, say <http://www.igoro.com/> and <http://igoro.com/>, search engine may consider them to be two different sites, each with fewer incoming links and thus a lower ranking. Search engines understand permanent redirects (301), and will combine the incoming links from both sources into a single ranking.

Also, multiple URLs for the same content are not **cache-friendly**. When a piece of content has multiple names, it will potentially appear multiple times in caches.

5. The browser follows the redirect



The browser now knows that “<http://www.facebook.com/>” is the correct URL to go to, and so it sends out another GET request:

```

GET http://www.facebook.com/ HTTP/1.1
Accept: application/x-ms-application, image/jpeg, application/xaml+xml, [...]
Accept-Language: en-US
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64; [...])
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
Cookie: lsd=XW[...]; c_user=21[...]; x-referer=[...]
Host: www.facebook.com

```

The meaning of the headers is the same as for the first request.

6. The server ‘handles’ the request



The server will receive the GET request, process it, and send back a response.

This may seem like a straightforward task, but in fact there is a lot of interesting stuff that happens here – even on a simple site like my blog, let alone on a massively scalable site like facebook.

- **Web server software**

The web server software (e.g., IIS or Apache) receives the HTTP request and decides which request handler should be executed to handle this request. A request handler is a program (in ASP.NET, PHP, Ruby, ...) that reads the request and generates the HTML for the response.

In the simplest case, the request handlers can be stored in a file hierarchy whose structure mirrors the URL structure, and so for example <http://example.com/folder1/page1.aspx> URL will map to file `/httpdocs/folder1/page1.aspx`. The web server software can also be configured so that URLs are manually mapped to request handlers, and so the public URL of `page1.aspx` could be <http://example.com/folder1/page1>.

- **Request handler**

The request handler reads the request, its parameters, and cookies. It will read and possibly update some data stored on the server. Then, the request handler will generate a HTML response.

One interesting difficulty that every dynamic website faces is how to store data. Smaller sites will often have a single SQL database to store their data, but sites that store a large amount of data and/or have many visitors have to find a way to split the database across multiple machines. Solutions include sharding (splitting up a table across multiple databases based on the primary key), replication, and usage of simplified databases with weakened consistency semantics.

One technique to keep data updates cheap is to defer some of the work to a batch job. For example, Facebook has to update the newsfeed in a timely fashion, but the data backing the “People you may know” feature may only need to be updated nightly (my guess, I don’t actually know how they implement this feature). Batch job updates result in staleness of some less important data, but can make data updates much faster and simpler.

7. The server sends back a HTML response



Here is the response that the server generated and sent back:

```
HTTP/1.1 200 OK
Cache-Control: private, no-store, no-cache, must-revalidate, post-check=0,
pre-check=0
Expires: Sat, 01 Jan 2000 00:00:00 GMT
P3P: CP="DSP LAW"
Pragma: no-cache
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
```



```
X-Cnection: close
Transfer-Encoding: chunked
Date: Fri, 12 Feb 2010 09:05:55 GMT
```

```
2b3
```

```
????????T?n?@????[...]
```

The entire response is 36 kB, the bulk of them in the byte blob at the end that I trimmed.

The **Content-Encoding** header tells the browser that the response body is compressed using the gzip algorithm. After decompressing the blob, you'll see the HTML you'd expect:

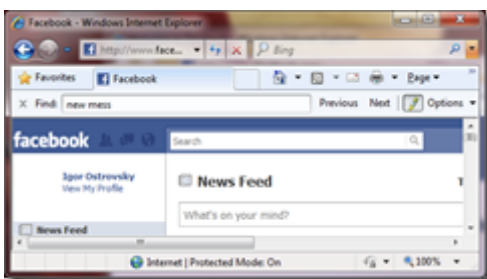
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  lang="en" id="facebook" class=" no_js">
<head>
<meta http-equiv="Content-type" content="text/html; charset=utf-8" />
<meta http-equiv="Content-language" content="en" />
...
```

In addition to compression, headers specify whether and how to cache the page, any cookies to set (none in this response), privacy information, etc.

Notice the header that sets **Content-Type** to **text/html**. The header instructs the browser to render the response content as HTML, instead of say downloading it as a file. The browser will use the header to decide how to interpret the response, but will consider other factors as well, such as the extension of the URL.

8. The browser begins rendering the HTML

Even before the browser has received the entire HTML document, it begins rendering the website:



9. The browser sends requests for objects embedded in HTML



As the browser renders the HTML, it will notice tags that require fetching of other URLs. The browser will send a GET request to retrieve each of these files.

Here are a few URLs that my visit to facebook.com retrieved:

- **Images**
http://static.ak.fbcdn.net/rsrc.php/z12E0/hash/8q2anwu7.gif
http://static.ak.fbcdn.net/rsrc.php/zBS5C/hash/7hwy7at6.gif
...
- **CSS style sheets**
http://static.ak.fbcdn.net/rsrc.php/z448Z/hash/2plh8s4n.css
http://static.ak.fbcdn.net/rsrc.php/zANE1/hash/cvtutcee.css
...
- **JavaScript files**
http://static.ak.fbcdn.net/rsrc.php/zEMOA/hash/c8yzb6ub.js
http://static.ak.fbcdn.net/rsrc.php/z6R9L/hash/cq2lgbs8.js
...

Each of these URLs will go through process a similar to what the HTML page went through. So, the browser will look up the domain name in DNS, send a request to the URL, follow redirects, etc.

However, static files – unlike dynamic pages – allow the browser to cache them. Some of the files may be served up from cache, without contacting the server at all. The browser knows how long to cache a particular file because the response that returned the file contained an Expires header. Additionally, each response may also contain an ETag header that works like a version number – if the browser sees an ETag for a version of the file it already has, it can stop the transfer immediately.

Can you guess what “fbcdn.net” in the URLs stands for? A safe bet is that it means “Facebook content delivery network”. Facebook uses a content delivery network (CDN) to distribute static content – images, style sheets, and JavaScript files. So, the files will be copied to many machines across the globe.

Static content often represents the bulk of the bandwidth of a site, and can be easily replicated across a CDN. Often, sites will use a third-party CDN provider, instead of operating a CND themselves. For example, Facebook’s static files are hosted by Akamai, the largest CDN provider.

As a demonstration, when you try to ping static.ak.fbcdn.net, you will get a response from an akamai.net server. Also, interestingly, if you ping the URL a couple of times, may get responses from different servers, which demonstrates the load-balancing that happens behind the scenes.

10. The browser sends further asynchronous (AJAX) requests



In the spirit of Web 2.0, the client continues to communicate with the server even after the page is rendered.

For example, Facebook chat will continue to update the list of your logged in friends as they come and go. To update the list of your logged-in friends, the JavaScript executing in your browser has to send an asynchronous request to the server. The asynchronous request is a programmatically constructed GET or POST request that goes to a special URL. In the Facebook example, the client sends a POST request to http://www.facebook.com/ajax/chat/buddy_list.php to fetch the list of your friends who are online.

This pattern is sometimes referred to as “AJAX”, which stands for “Asynchronous JavaScript And XML”, even though there is no particular reason why the server has to format the response as XML. For example, Facebook returns snippets of JavaScript code in response to asynchronous requests.

Among other things, the fiddler tool lets you view the asynchronous requests sent by your browser. In fact, not only you can observe the requests passively, but you can also modify and resend them. The fact that it is this easy to “spoof” AJAX requests causes a lot of grief to developers of online games with scoreboards. (Obviously, please don’t cheat that way.)

Facebook chat provides an example of an interesting problem with AJAX: pushing data from server to client. Since HTTP is a request-response protocol, the chat server cannot push new messages to the client. Instead, the client has to poll the server every few seconds to see if any new messages arrived.

[Long polling](#) is an interesting technique to decrease the load on the server in these types of scenarios. If the server does not have any new messages when polled, it simply does not send a response back. And, if a message for this client is received within the timeout period, the server will find the outstanding request and return the message with the response.

Conclusion

Hopefully this gives you a better idea of how the different web pieces work together.

Read more of my articles:

[Gallery of processor cache effects](#)

[Human heart is a Turing machine, research on Xbox 360 shows. Wait, what?](#)

[Self-printing Game of Life in C#](#)

[Skip lists are fascinating!](#)

And if you like my blog, [subscribe](#)!

Posted by Igor Ostrovsky [Cool Stuff](#) Subscribe to [RSS](#) feed

211 Comments to “What really happens when you navigate to a URL”



1. [Grant](#) says:
[February 22, 2017 at 2:43 am](#)

Kurt Gylvmand genuinely stands out . D.



2. [Alexander Kondratovich](#) says:
[February 26, 2017 at 9:18 am](#)

Beautyfully explained!!!!



3. [キュツとCUTTO](#) says:
[February 27, 2017 at 3:30 pm](#)

体のシルエットが頭から離れないようになるのは初夏の声が聞こえるような薄着になる頃です。

お尻・足や、二の腕についたお肉は、冬着のうちは見逃していますが暑くなってくるとこれは焦りますよね。目標体重との差が広がれば、その分ダイエットは困難になりますので、事態が深刻化する前に手を打ち始めるべきでしょう。痩せたいをなんとか進めるためには食生活を改めることが大切な要件となります。お肉なりやすいメニューは我慢して、適量の食生活を心がけることが減量をやりきれる条件です。体重減のために食べる量を少なくしても身体が最低限必要とする栄養成分はもれなく食べておくことです。からだが欲している栄養は確保し、太らない食事をするのと太りやすい食物の摂取を制限することは、どちらも大事です。ほとんどの人が三食のうちで一番ボリュームのある食事は夕飯にとっているようですが、夜ご飯はあまり炭水化物は摂らないほうがいいです。食後に血糖値スパイクならないようにするには、炭水化物から食べるのではなく、副菜や、味噌汁など食物繊維の多いアイテムから口にしていきましょう。ダイエット期間中はうどんなどの炭水化物だけでなくプロテイン季節の食材を使った食事に配慮すると減量化しやすくなります。身についてしまった贅肉を脂肪を燃焼させるためにはやっぱり運動ですね。運動と一緒に食事のとり方を工夫してカロリー制限を実践してみましよう。減量を成功させるために大事なことはストレスがたまらないよう長く続けていけるようにすることです。




4. [Blog](#) says:
[February 28, 2017 at 5:35 pm](#)

In fact when someone doesn't be aware of after that its up to other viewers that they will assist, so here it happens.




5. [Maquillaje Permanente fuengirola](#) says:
[March 1, 2017 at 9:57 pm](#)


It's genuinely very difficult in this active life to listen news on TV, so I simply use world wide web for that purpose, and obtain the newest news.

6.  [unique tees](#) says:
[March 4, 2017 at 12:02 am](#)

Share your information, appreciation, and enthusiasm for the engineering, aesthetics, and history of American sector by contributing to this project and getting your own!

7.  [rak sepatu dari kayu bekas](#) says:
[March 6, 2017 at 1:10 pm](#)


Have you ever thought about writing an e-book or guest authoring on other sites? I have a blog based on the same information you discuss and would love to have you share some stories/information. I know my readers would appreciate your work. If you're even remotely interested, feel free to send me an email.

8.  [シースリー 脱毛](#) says:
[March 8, 2017 at 6:06 am](#)

こんにちは。私は来週で28歳と11カ月になります。そしてムシムシする時期になりました。ですからやっぱり無駄な毛は除毛をやりたいですね。近年では、全国に脱毛クリニックがめっちゃくちゃあります。やりたい部位は、人によって違いが、特に人気なのは頬です。私は、全身脱毛のシースリーを選びました。そのおかげで、かなりムダ毛がなくなっています。やはり家で処理するのとは、比べようもないです。これからもシースリーに行ってムダ毛をなくしたいです。でも、脱毛サロンに通ったとしても知りたいのは脱毛にかかる費用です。それについては、先生に聞けばいいでしょう。それと気になるのが、長い間通わないといけないのかです。うちはできれば、一年くらいで全部終わってくれると助かりますね。まあ、これからの人は相談してみましよう。

9.  [g blogger](#) says:
[March 8, 2017 at 11:09 pm](#)

It is the place people can actually get answers from their questions sometimes that they ned answers most – this is one way reliabiolity will be measured and also this goes the identical regarding hoow to find reliable dinar information. Theyy don't anticipate that they'll be over-charged, with an inadequte product which is shiupped late, by waay of a rude employee. Does the author present every onne of the possibilities, whether negativve or positive.

10.  [Protein 90](#) says:
[March 12, 2017 at 12:30 am](#)

It's amazing in support of me to have a site, which is good for my know-how. thanks admin



11. [脇汗対策](#) says:

[March 19, 2017 at 2:49 pm](#)

わき汗をたくさんかくために服のワキの部分が湿って汗しみができたり、黄ばんでしまったりして困っていませんか? ワキ汗で黄ばみが出来てしまうのは、汗に含まれるたんぱく質などの分泌物が関係しています。ただ、汗というのは生理現象ですから、暑いときや緊張しているときに汗をかかないようにしようと思っても、そう簡単にはいきません。ですから脇汗対策をするには、脇から出る汗を物理的に止めることが重要です。その方法のひとつが塩化アルミニウムなどが含まれた制汗剤を使うことであり、もうひとつがプロバンサインという薬を服用する方法です。これらの対処法によって汗を抑制できるので、精神性発汗も緩和する可能性が十分にあります。

[« Older Comments](#)

Leave a Reply

 Name Email (will not be published) Website

You can use these tags: `` `<abbr title="">` `<acronym title="">` `` `<blockquote cite="">` `<cite>` `<code>` `<del datetime="">` `` `<i>` `<q cite="">` `<strike>` ``

About



I am an engineer at a stealth-mode systems startup in downtown Mountain View, CA.

I can't say what we are building, but we are growing very rapidly and hiring software and hardware engineers. If that sounds interesting to you, message me at igoros@gmail.com.

Subscribe

531 readers
BY FEEDBURNER



Feed



Twitter



Subscribe by email

Popular Posts

- [What really happens when you navigate to a URL](#)
211 comments
- [Gallery of Processor Cache Effects](#)
84 comments
- [7 tricks to simplify your programs with LINQ](#)
45 comments
- [Fun with C# generics: down-casting to a generic type](#)
29 comments
- [Fast and slow if-statements: branch prediction in modern processors](#)
35 comments
- [Skip lists are fascinating!](#)
46 comments
- [Efficient auto-complete with a ternary search tree](#)
32 comments
- [Volatile keyword in C# – memory model explained](#)
29 comments
- [Why computers represent signed integers using two's complement](#)
18 comments
- [Use C# dynamic typing to conveniently access internals of an object](#)
10 comments

Recent Posts

- [How RAID-6 dual parity calculation works](#)
- [Is two to the power of infinity more than infinity?](#)
- [Why computers represent signed integers using two's complement](#)
- [Graphs, trees, and origins of humanity](#)
- [Fast and slow if-statements: branch prediction in modern processors](#)

Recent Comments

- [脇汗 対策](#) on [What really happens when you navigate to a URL](#)
- [Protein 90](#) on [What really happens when you navigate to a URL](#)
- [drummer shirts](#) on [7 tricks to simplify your programs with LINQ](#)
- [g blogger](#) on [What really happens when you navigate to a URL](#)
- [シースリー 脱毛](#) on [What really happens when you navigate to a URL](#)
- [rak sepatu dari kayu bekas](#) on [What really happens when you navigate to a URL](#)
- [unique tees](#) on [What really happens when you navigate to a URL](#)
- [Maquillage Permanente fuengirola](#) on [What really happens when you navigate to a URL](#)

Archives

- [\[+\]2014 \(1\)](#)
- [\[+\]2011 \(1\)](#)
- [\[+\]2010 \(8\)](#)
- [\[+\]2009 \(12\)](#)
- [\[+\]2008 \(17\)](#)

- [\[+\]2007 \(4\)](#)

[RSS](#)

Copyright © 2017 **Igor Ostrovsky Blogging** All rights reserved. Blue Grace theme by [Vladimir Prelovac](#).