Day 3

This is CS50 for MBAs. Harvard Busines School. Spring 2015.

Cheng Gong

Table of Contents

Announcements

- On tonight's assignment, we'll be asking a question about how long you'd like class to be, now that you have an idea of how dense the material is and how long two hours feels like, so please provide us feedback!
- For tomorrow, we'll try to be done around 2:35pm, as we introduce some web development (HTML and CSS) and a web-based IDE (integrated development environment) that lets you use a real server!

Data Types

- A note from a student, who works part time as a Product Manager, happened to talk to some engineers the other day about storage. They talked about how a "float", a data type for storing floating-point values, is not efficient for storing prices. They actually use a "bigint", big integer, and just divide by 100 to represent an actual price. So this is a good opportunity to talk about data types.
- A computer has a finite number of bits in memory. And there are infinitely many real numbers, so we might run into a problem with precision, since we can't possibly represent all numbers with a limited amount of bits.
- Let's take a look at this program, written in C:

```
#include <stdio.h>

int main(void)
{
    float f = 1.0 / 10.0;
    printf("%.1f\n", f);
}
```

If we look inside the braces, we notice a `float f` to the left of an `=`. This just says that we want a variable with enough space for a `float`, called `f`, in which we can store a value like `1.2` or `1.23`. On the right, we're doing some math that stores `1.0 / 10.0` into our variable called `f`.

In the next line, we're just printing out `f`, with the `%.1` telling the computer to show 1 decimal place, and `\n` saying we want a new line after the number is printed.

- So in theory, this should print `0.1`, and after we compile and run it, we do see `0.1`.

Imprecision and Overflow

- But let's look further and see not 1 but, say, 28 places after the decimal point:

```
#include <stdio.h>

int main(void)
{
    float f = 1.0 / 10.0;
    printf("%.28f\n", f);
}
```

- We expect something like `0.100000000000…` but we get:

```
0.1000000014901161193847656250
```

- Wait, what?! Turns out, the computer is storing the closest match to `0.1` that a computer has to represent a float.
- The number of bits used for certain data types have generally been standardized. Here are the sizes of some types used in a programming language called Java:

```
bits type
```

```
1    boolean (either true or false, so only one bit is needed)
8    byte (by definition)
16   char (character, 8 bits in ASCII, but 16 bits in Unicode)
16   short (for a short number, largest number being 2^16 - 1, roughly
 65,000)
32   int
32   float (which means there are only ~4 billion floating point values we
 can actually represent)
64   long (longer integer)
64   double (float with twice as many bits for more precision)
```

- We can avoid this problem in the financial world by storing integers, which are exact. We can do this by storing values in cents rather than dollars, avoiding the need for floats.
- Though this seems insignificant, in software where we add and multiply and use numbers over and over again, these tiny mistakes add up.
- Another problem we might encounter is **integer overflow**.
- Let's look at this number, where there are 8 bits, and all of them are 1:

```
128   64   32   16   8    4    2    1   (place value)
 1    1    1    1    1    1    1    1   (bits)
```

- If we calculated the value of this number, we'd get 255. But if we added 1 to this number, we'd expect something like this to represent 256:

```
    128   64   32   16   8    4    2    1   (place value)
1    0    0    0    0    0    0    0    0   (bits)
```

- But since we only have 8 bits, we can only store the last 8 digits, and get a value of 0 instead of 256:

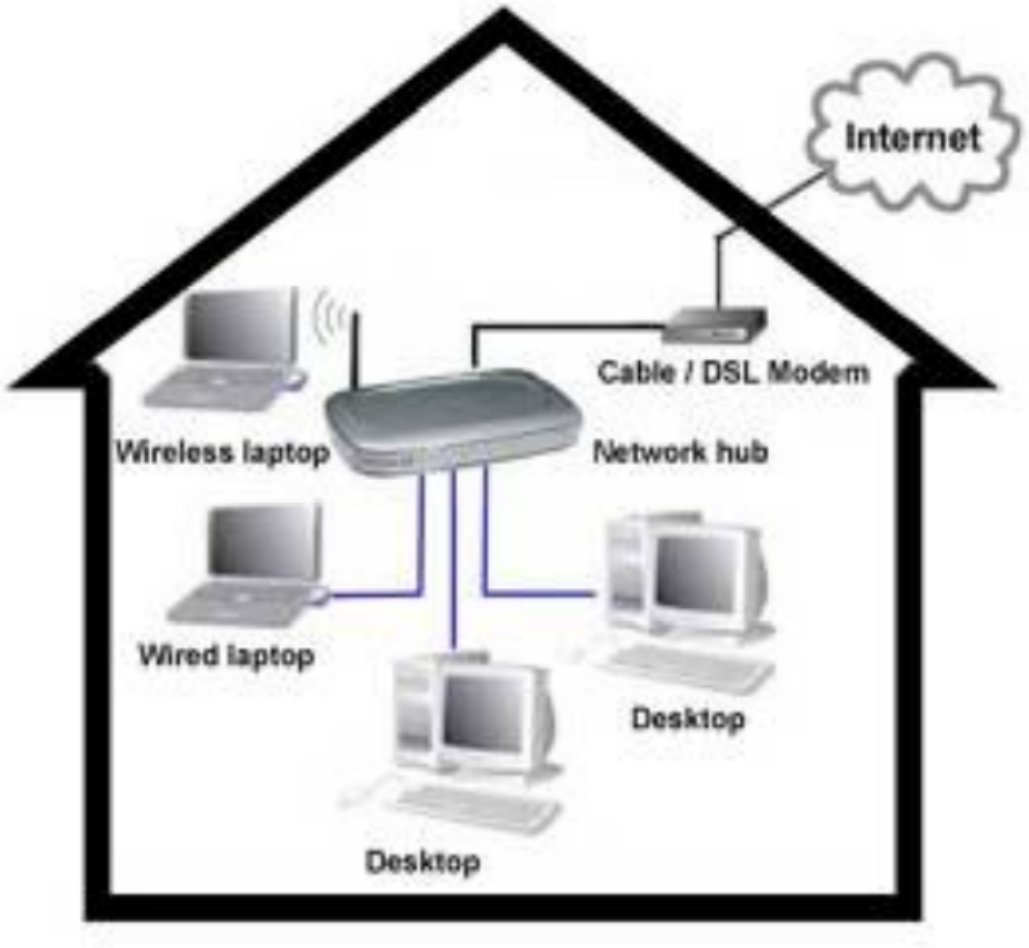
```
128   64   32   16   8    4    2    1   (place value)
 0    0    0    0    0    0    0    0   (bits)
```

Internet

- We watch a few clips from TV shows where fancy technical terms are jumbled together, but don't actually mean anything.
- In one of the clips, a screen has the following in the address bar:

```
http://275.3.6.28
```

- This looks like a real IP address, but isn't. (Almost) every computer on the Internet has an IP address, to identify its location, and is a set of 4 bytes. So each number has 8 bits, which means that the `275` in the screenshot isn't in the range of real IP addresses.
- But when we go on the Internet, we don't normally type in these numbers. Instead, we have domain names like `www.facebook.com` that maps to these numbers.
- In a typical home, we might find the following:



We have some kind of Internet service (the cloud outside) that we connect to with a cable or DSL modem that the Internet service provider (ISP) gives us, with one wire that plugs into the wall and outside and another wire that goes into a router (or network hub) that your computers actually connect to, with WiFi or ethernet.

Some ISPs give you an all-in-one device that is both a modem and a router.

- But how do we actually connect to the Internet? Our ISP gives us an IP address, which we need to share with multiple computers somehow, and we also need to find the IP address of a server just by typing in a domain name.
- So there's a Domain Name System, **DNS**, that keeps track of these conversions. When someone buys a domain name, they tell some central authority what their IP address is, and ISPs like Comcast will have copies of the DNS database that match IP addresses to domain names.
- And there's a hierarchy as a result of **caching**, where IP addresses you looked up recently are stored nearby (whether on your laptop, the router, or the ISP's DNS server) for some amount of time, so it doesn't take forever to look them up again.
- Now that we have the IP address of the server, we need our own so that they can send us information back.
- Dynamic Host Configuration Protocol (**DHCP**) is the fancy term for how computers acquire IP addresses. Essentially, there are servers that your computer can talk to, to ask for an IP address.
- Let's put this a little more concretely. David has a printed picture of a cat that he wants to send to Dan, who's at the back of the room.
- Since picture files are pretty big, we'll tear the picture and send it as chunks.
- David puts each chunk into an envelope, with his name in the From: field and Dan's name in the To: field.
- He passes the envelopes to people nearby, who represent routers, and eventually the envelopes all get to Dan.
- On the Internet, each number in an IP address tells every server in between which direction to send the packet (like an envelope) to, and eventually they'll get to the person they were intended for.
- There are also lots of paths that could be taken, that would eventually reach the destination, so if some servers went down data could still be sent.
- Some routers on the Internet could also get too many packets and once, and that means some will get dropped.
- The envelopes, too, were numbered 1 out of 4, 2 out of 4, etc., so even if they arrived out of order the person at the other end can reassemble the picture or message correctly.
- We can actually run a program in our command line called `traceroute` that tells us the route our packets took:

```
% traceroute -q 1 www.mit.edu
traceroute to www.mit.edu (172.230.86.119), 30 hops max, 40 byte packets
 1  mr-sc-1-gw-vl-3085-fas.net.harvard.edu (140.247.89.129)  0.673 ms
 2  10.240.143.25 (10.240.143.25)  1.053 ms
 3  coregw1-vl-416-fas.net.harvard.edu (140.247.2.65)  0.842 ms
 4  bdrgw1-te-4-2-core.net.harvard.edu (128.103.0.18)  1.205 ms
 5  bst-edge-05.inet.qwest.net (63.145.1.133)  9.059 ms
 6  nyc-edge-04.inet.qwest.net (205.171.30.62)  6.854 ms
 7  a172-230-86-119.deploy.static.akamaitechnologies.com (172.230.86.119)
 6.998 ms
```

- That was pretty fast, with `www.mit.edu` just 7 physical routers away.
- It looks like `www.mit.edu` was translated with DNS to an IP address first, `172.230.86.119`, and we see that the first place we sent our packet to looks like server from Harvard FAS, with `mr` and `sc` referring to Science Center Machine Room, and `gw` meaning "gateway."
- Then the next one doesn't have a name, and the third is another Harvard FAS server, which seems to be a more important "core" gateway.
- The fourth one is probably a "border" gateway, maybe toward the outside of Harvard's network, and it looks like that connects to an ISP called Qwest, first to a server in Boston, and then in NYC. Then it goes to a server owned by Akamai Technologies.
- So it seems that MIT's website aren't on servers near them, but rather somewhere in the cloud, hosted by a third party.
- We can run another command:

```
% whois mit.edu
...
Domain Name: MIT.EDU

Registrant:
   Massachusetts Institute of Technology
   77 Massachusetts Ave
   Cambridge, MA 02139
   UNITED STATES

Administrative Contact:
   Mark Silis
   Massachusetts Institute of Technology
```

```
   MIT Room W92-167, 77 Massachusetts Avenue
   Cambridge, MA 02139-4307
   UNITED STATES
   (617) 324-5900
   mark@mit.edu

Technical Contact:
   MIT Network Operations
   Massachusetts Institute of Technology
   MIT Room W92-167, 77 Massachusetts Avenue
   Cambridge, MA 02139-4307
   UNITED STATES
   (617) 253-8400
   noc@mit.edu

Name Servers:
   USE2.AKAM.NET
   USE5.AKAM.NET
   USW2.AKAM.NET
   ASIA1.AKAM.NET
   ASIA2.AKAM.NET
   EUR5.AKAM.NET
   NS1-173.AKAM.NET
   NS1-37.AKAM.NET

Domain record activated:    23-May-1985
Domain record last updated: 17-May-2013
Domain expires:             31-Jul-2015
```

- This command, `whois`, asks the root servers, or the central, highest authority on who owns which domain names.
- These root servers actually don't know the actual IP addresses, but rather points to name servers that would know. In MIT's case, since their site is hosted by a company called Akamai, the name server records point to those servers.
- We can run this command again:

```
% whois google.com
...
Name Server: ns1.google.com
Name Server: ns2.google.com
Name Server: ns3.google.com
Name Server: ns4.google.com
```

. . .

- So Google has its own DNS servers,
- If a website changes web hosts or servers, it might be some time before that change is reflected everywhere, so some people might still get an outdated IP address that no longer works. To minimize this, we might configure the "expiration date," or TTL (time to live) of our domain name to be a shorter time. That way, when the information is updated, servers in between will only cache the old information for a small amount of time before it gets the new information.
- DNS poisoning is an attack where a country, or some group in control of those central servers, returns the wrong answer to DNS lookups, and redirects traffic to their servers, or taking entire websites offline by returning something like "site not found."
- Some ISPs would also do this, but only for invalid URLs that had no legitimate entries. Instead, they would take you to pages with advertisements somehow related to that invalid URL.
- If we go to the actual IP address of some domain name, it should generally bring us to the website (unless multiple websites are sharing the same server with the same IP address, in which case we need to ask it for the website we want in a more sophisticated way).
- DNS entries are cached so future visits are faster, even if we risk having outdated entries for some amount of time.
- We can also do something like this:

```
% traceroute -q 1 www.cnn.co.jp
traceroute to www.cnn.co.jp (103.4.202.6), 30 hops max, 40 byte packets
 1  mr-sc-1-gw-vl-3085-fas.net.harvard.edu (140.247.89.129)  0.838 ms
 2  10.240.143.25 (10.240.143.25)  1.181 ms
 3  coregw2-te-8-4-core.net.harvard.edu (140.247.2.37)  1.206 ms
 4  bdrgw2-te-4-2-core.net.harvard.edu (128.103.0.2)  1.670 ms
 5  bst-edge-05.inet.qwest.net (63.145.1.133)  1.357 ms
 6  pax-brdr-01.inet.qwest.net (205.171.234.102)  70.169 ms
 7  ae-4-90.edge1.SanJose2.Level3.net (4.69.152.199)  72.607 ms
 8  pajbb002.int-gw.kddi.ne.jp (111.87.3.37)  71.403 ms
 9  pajbb002.int-gw.kddi.ne.jp (111.87.3.101)  73.096 ms
10  tm4BBAC01.bb.kddi.ne.jp (106.162.175.26)  176.763 ms
11  182.248.216.222 (182.248.216.222)  171.354 ms
12  14.0.32.62 (14.0.32.62)  168.432 ms
13  103.4.202.6 (103.4.202.6)  174.038 ms
```

- Here we're connecting to Japan's edition of CNN. That was still pretty fast, but we do see more routers in between, and longer reponses times in the hundreds of milliseconds.
- Router 7 in particular seems interesting, it looks like our packet passed through San Jose, and in between router 9 and 10 we seem to have crossed the Pacific Ocean, since the response time increased so much. Routers 8 and 9 ended in `.jp`, but the company that owned them might just have some US servers.
- There are all of these hops because we just don't have that many direct connections. (It's sort of like how railroad tracks go from major city to major city, and even though we can't possibly lay down tracks between every pair of cities, there's always a connection we can take to get between them.)
- CS50 actually recently signed up for an Amazon service called Direct Connect[1] that allows our servers to have a more direct connection, with fewer hops, to Amazon's cloud servers, so we get better performance, at a cost of money.
- Net neutrality is related, as there is a question (in simple terms) of whether ISPs should be allowed, while routing, to give priority to packets from, say, Skype, or Google Hangouts, if those companies are paying them.
- And typically, the size of a packet is something like 1000 to 2000 bytes, although the maximum size can be configurable within each individual network.
- It happens to be that certain companies or geographies get IP addresses, and even though the boundaries aren't strict these days with so many mobile devices, there are still some loose assignments.
- For example, Harvard's pool of IPs start with the following:

```
140.247.#.#
128.103.#.#
...
```

- Private IPs are reserved for devices on your local network,a nd not accessible by the Internet at large:

```
10.#.#.#
172.16.#.# – 172.31.#.#
192.168.#.#
```

[1] http://aws.amazon.com/directconnect/

- This helps solve the problem of sharing one public IP address among the multiple devices in your home, with a technology called NAT, network address translation.
- Your router keeps track of which packets are from which device internally, even though they are all sent out and come back in with the same public IP address.
- If we think about how many bits are in an IP address, there are only 32, meaning that there are only 4 billion total, with some reserved for private addresses. So when we run out, we'll have to switch to a new version called IPv6 with 128 bit addresses. DNS servers can already return these longer addresses, but only some websites have adopted it.
- Companies and universities might also give out private IPs to users within the network, so they only take up one public IP.

TCP, Ports, HTTP

- There are lots of protocols in networking to make everything standard and working correctly, but a particular noteworthy one is **TCP**, Transmission Control Protocol, which is responsible for getting all the packets to their destination. For example, with our cat picture, which was split into 4 chunks, if the recipient only got 3 of them, they could look at the outside of the envelope, since they are labeled "1 of 4", "2 of 4", etc, and tell the sender which ones they got and which are missing.
- Another one protocol is UDP, where we don't ask for acknowledgement, and just send out packets or envelopes and hope they get there. We'd use that for things like live streaming video, where we want as much of the data to get there as possible, but if we lost a few, then it's too late anyways by the time we resent them.
- We also want computers on the internet to do lots of types of things, like email or file transfers. A web server is just a piece of software, that can live on even your laptop, that communicates with the network and returns information in some way.
- Servers know what certain types of packets are for, because packets all happen to have a number called a **port** on them in addition to the IP address:

```
ports
21 FTP
25 SMTP
53 DNS
80 HTTP
443 HTTPS
```

    # Port `80` is meant for HTTP, or web browsing, meaning data that comes into or out of that port number is most likely data for a webpage.
    # Port `25` will indicate that the packet is for email, and so on.

- There are lots of these such port numbers, so servers can perform many different functions.
- We'll focus now on **HTTP**, Hypertext Transfer Protocol. A protocol is just a standard, like giving people handshakes when you meet them. (The term handshake is actually used in computing too, to indicate a two-sided communication.)
- HTTP is just a standard for how web servers and clients communicate, and what commands they might send each other.
- Hypertext is interesting too, since it refers to hypertext markup language (better known as HTML) that we use to transmit text and images and links, and we'll go more into that next week.
- Let's take a look at this (somewhat outdated) picture:

- There is a browser running on the left and a server on the right, and the browser actually makes standard requests to the server to get webpages and other information.
- One of these request protocols is called **GET**. The browser will send a packet with IP addresses on the outside like before, but with text like this on the inside:

```
GET / HTTP/1.1
Host: www.google.com
```

```
...
```

- On the first line, `/` indicates the file it wants, in this case just the default page and `HTTP/1.1` just refers to the version of HTTP the browser supports.
- On the second line, we're telling the server that we want this website, just to be sure we get the right one, since on the outside of the packet is just an IP address, and there might be multiple web servers on the same IP address.
- In response, we hope to get something like this back:

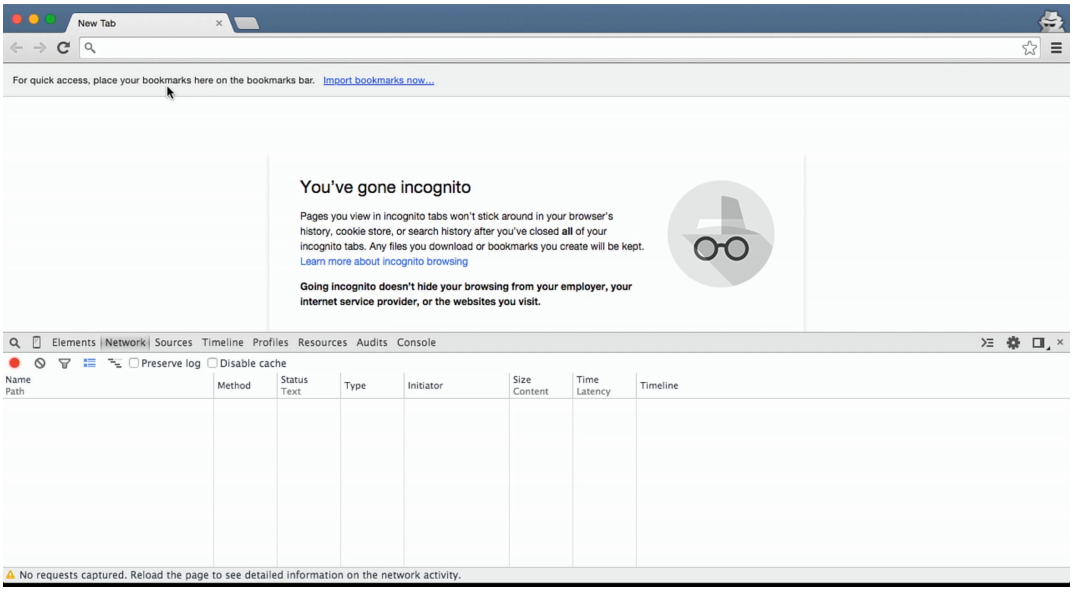
```
HTTP/1.1 200 OK
Content-Type: text/html
...
```

… with the actual content of the page following it in the `…`.

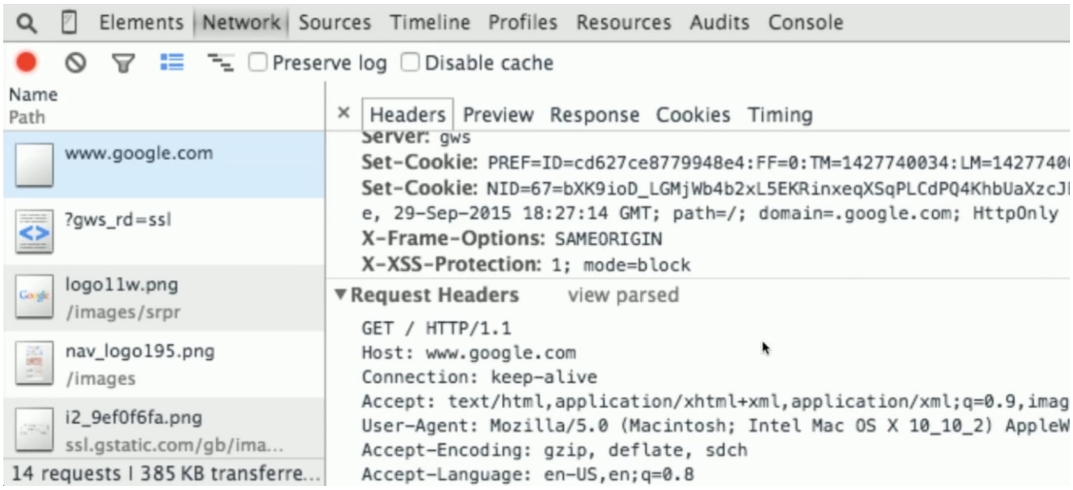
- If we wanted to do a search, the request might look like this:

```
GET /search?q=cats HTTP/1.1
Host: www.google.com
...
```

- We'll take a look at this in a second, but first realize that Chrome has this Developer Tools feature, that you can open with View > Developer > Developer Tools:



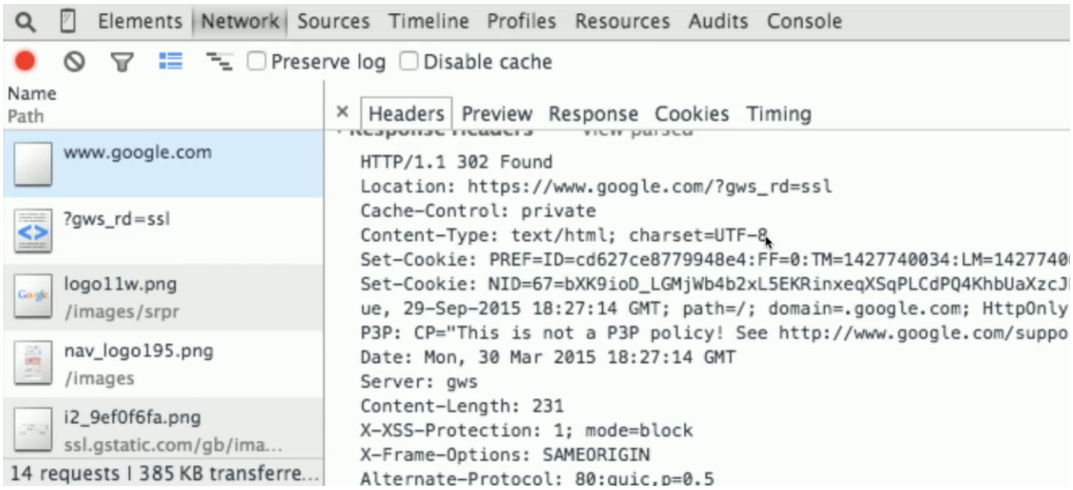
- We'll switch to the Network tab, make sure the recording icon on the left is on, and visit `http://www.google.com`:



We scrolled up in the column on the left until we saw `www.google.com`, and then scrolled down in the column to the right until we saw `Request Headers`.

That's what our browser actually sent to Google, and the other things below are somewhat useful but not essential.

- If we scroll up a bit in the right column:



… we see that the response headers say `302 Found`. `302 Found` and `200 OK` are two of many codes, out of which `404 File Not Found` being a common one that you might have seen before.

Notice that it's telling us the `Location` is actually `https://...`, meaning that the server is redirecting us to the secure version of `google.com`.

```
# Then there are a whole bunch of other things we get back too, that we'll take a closer look at later.
```

- We can also use another, much older program called `telnet` to try to talk to web servers:

```
% telnet www.google.com 80
Trying 4.53.56.118...
Connected to www.google.com.
Escape character is '^]'.
```

- Now we can type as though we were a browser, so we add:

```
GET / HTTP/1.1
Host: www.google.com
```

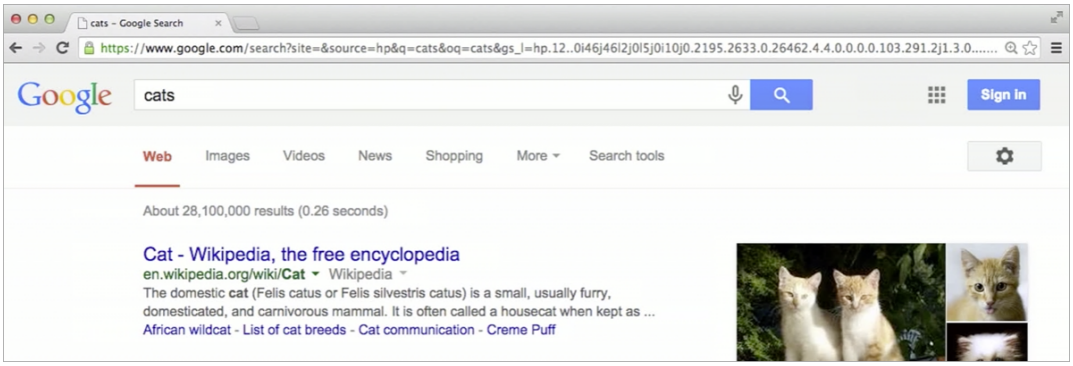
- After hitting enter, we get a lot of code, but if we scroll up to the beginning, we see:

```
HTTP/1.1 200 OK
Date: Mon, 30 Mar 2015 18:29:34 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie:
 PREF=ID=4e7f2fcb244abe54:FF=0:TM=1427858846:LM=1427858846:S=_gdpkzF2ApXil9Z0;
 expires=Fri, 31-Mar-2017 03:27:26 GMT; path=/; domain=.google.com
Set-Cookie:
 NID=67=q1JfI1_IlEQq7CPuXb_Fdi6SlKTCyFDEnHW5Y9zWdNbYLQBoYeE0jYld4EjtUJo54etAmCPhR4TVLsJ
 expires=Thu, 01-Oct-2015 03:27:26 GMT; path=/; domain=.google.com;
 HttpOnly
P3P: CP="This is not a P3P policy! See http://www.google.com/support/
accounts/bin/answer.py?hl=en&answer=151657 for more info."
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Alternate-Protocol: 80:quic,p=0.5
Accept-Ranges: none
Vary: Accept-Encoding
Transfer-Encoding: chunked

2572
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage"
 lang="en"><head><meta content="Search the world's information, including
```

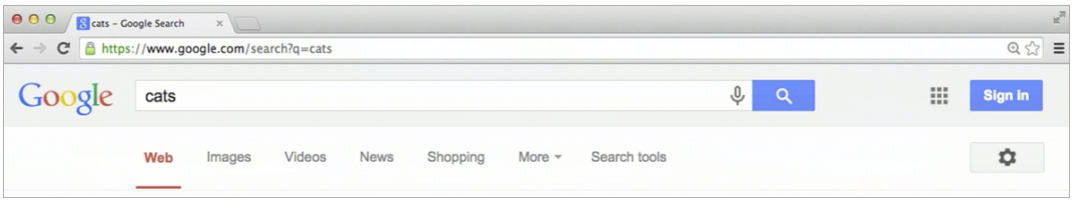
```
  webpages, images, videos and more. Google has many special features to
  help you find exactly what you're looking for." name="description"><meta
  content="noodp"
...
```

- If we scroll down, we see lots of code that a browser would render as Google's homepage, but notice that last line with tags that start with `<html`, indicating that the text is actually in HTML.
- If we go back to Google in our browser, and type in `cats`, we see this in the address bar:



  # There are lots of things there now, but in particular we see `/search?` followed by a pattern of `???=???` with `&` between each of them. (That first `?` after `search` is giving the page some input, rather than ask for a page with that specific name.)

- We'll get rid of all the stuff we don't understand, but keep this:



  # And it still works.

- But realize that our input is being sent in the URL, for anyone looking over your shoulder to see, or in the history later.
- An alternative to **GET** is a protocol called **POST**, which sends a request not in the URL but later in the request:

```
POST /login.php HTTP/1.1
```

```
Host: www.facebook.com
...
email=jharvard@cs50.harvard.edu&password=crimson
```

- POST also allow us to upload files like photos, since we can't fit an entire image into a URL, but the rest of the request can be bigger.
- Here are some more status codes:

```
200 OK
301 Moved Permanently
302 Found
401 Unauthorized
403 Forbidden
404 Not Found
500 Internal Server Error
...
```

- Another part of the response header you might have noticed is something called a **cookie**. In a response, it might look something like this:

```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: session=abc123; HttpOnly; Path=/
...
```

And all it does is save a piece of information from a server on a computer, like `abc123` (though in reality it's probably a much longer string of letters and numbers).

- You can think of a cookie like a digital handstamp, like one you might get at an amusement park, so later they can identify who you are more easily.
- Indeed, later when our browser makes another request, it sends our cookie back to the server:

```
GET / HTTP/1.1
Host: www.facebook.com
Cookie: session=abc123
```

And that's how we stay logged in to websites like Facebook without having to enter our username and password every time.

- So if our browser is sending these requests without encryption, as in using HTTP instead of HTTPS, then someone could theoretically capture that handstamp and pretend to be you, in an attack called **session hijacking**.
- A more sophisticaed web server might avoid this issue by checking the IP address in addition to the handstamp, but what if I move my laptop and connect to WiFi somewhere else, getting another IP address?
- A few years ago this was more common, but these days most websites do require HTTPS, where everything, including requests and response headers, are encrypted.
- To be continued!