Intro to the Internet

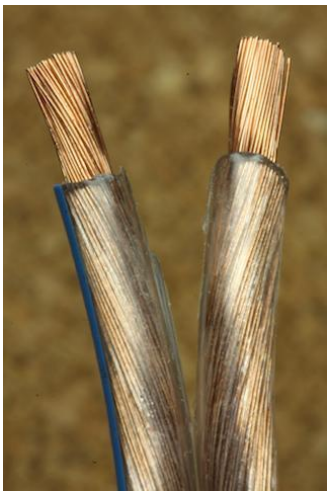
Prerequisites

- Intro to Computer Hardware and Performance
- Command Line

What you'll learn

- How the Internet works.
- HTTP
- REST
- Cookies

Turing on a Wire



How do you send the a Turing tape across a wire? It's actually pretty straightforward. 1 represents a high voltage. 0 represents a low voltage. When you want to send a 1, you apply a high voltage to one end of the wire, and it's detected on the other end. When you want to send a zero, you apply a low voltage to one end.

[demo this?]

The real question is: how do you let another computer know when a series of 0s and 1s begins and ends? Should it just wait? If so, how long? What we need is an official procedure for determining these things. What we need is a protocol.

IP and IP Addresses

Format: NNN.NNN.NNN.NNN

What's my ip address?
http://remoteip.me/

Data transmission over the internet is split into two parts. The first, called IP or "Internet Protocol", allows my computer to locate any other computer connected to the internet. Everything connected to the internet is assigned a unique number, called it's IP address. When written out for humans, an IP address is represented by 4 numbers, ranging from 0 to 256, separated by periods.

There are a number of websites, like remoteip.me and ipchicken.com, that will tell you what your IP address is.

To connect to a computer, I first broadcast a message to everyone on the network asking if anyone knows where to find the computer at some address. I don't have a direct connection to every computer on the network, so some computers will forward my request along to other networks they know to help me find what I'm looking for.

Localhost

The special IP Address 127.0.0.1, also known as localhost or the loopback address, points right back at you. It allows you to access your own computer as if you were accessing it over the internet.

DNS

```
nslookup google.com

Address: 74.125.21.113
Address: 74.125.21.101
Address: 74.125.21.100
Address: 74.125.21.139
Address: 74.125.21.102
Address: 74.125.21.138
```

How do I find a computer's IP address if I don't already know it?

The Internet's Domain Name System (DNS) is a global phonebook that turns text domain names into IP Addresses. I can lookup the ip addresses associated with any domain using the "nslookup" command in Terminal. Google.com, for example, has 6 ip addresses. I can connect to any of these 6 addresses to access the servers running the Google search engine.

Port

206.190.36.45:12345 ⇔ 74.125.21.138:54321

There are 2^{16} (65536) possible ports.

Once I've found the computer I'm looking for, I'm assigned a port. All further communication is direct, one computer to the other. There are no more worldwide broadcasts. My ip-address and port talks directly to another computer's ip-address and port.

Every computer has 2^{16} or 65536 possible ports. That means every computer can connect to at most 65,000 other computers. In reality, you'll never get close to that number. You computer's hardware will be overwhelmed managing all those conversations way before then.

TCP

TCP pseudo-header for checksum computation (IPv4)

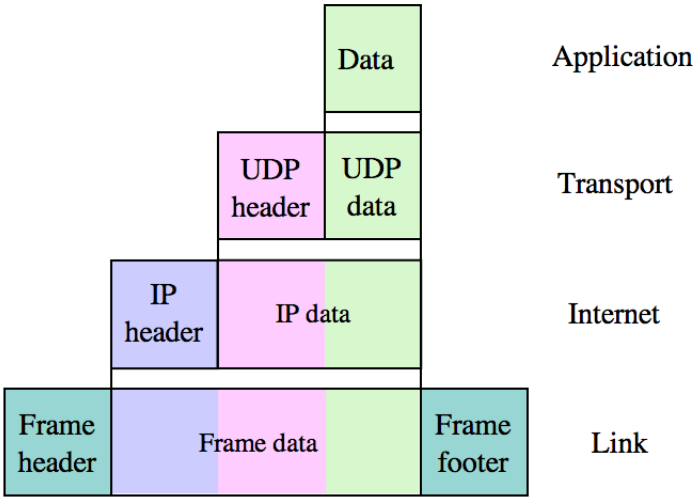
Bit offset	0–3	4–7	8–15	16–31
0	Source address			
32	Destination address			
64	Zeros		Protocol	TCP length
96	Source port			Destination port
128	Sequence number			
160	Acknowledgement number			
192	Data offset	Reserved	Flags	Window
224	Checksum			Urgent pointer
256	Options (optional)			
256/288+	Data			

Once I've found the computer I'm looking for and established a port-to-port connection, I switch to the TCP (Transmission Control Protocol) to communicate.

IP is fast, but unreliable. I don't care who responds to my address search. Most computers won't get the message at all or, if they do, will ignore it.

TCP is slower, but more reliable. When it's time to have a serious conversation with one other computer, it's time to switch to TCP.

TCP/IP Packets



In both of the Internet protocols, TCP and IP, the Turing tape data I'm trying to transmit is broken up into chunks called packets. Each packet contains a header (some helpful data at the beginning), an optional footer (some helpful data at the end), and a payload (everything in the middle).

Breaking my message up into chunks makes the transmission more reliable. If something goes wrong, I only have to retransmit the packets that were broken or lost instead of the entire message.

This is also the reason why the guy next to me can watch a movie on his computer while I'm still able to check my email. He gets to send/receive a packet, then I get to send/receive a packet, and so on. Everyone takes turns using the network.

HTTP

Once I've established my connection, the TCP fades into the background and it's time to talk business. TCP is still being used in the background, sending packets back and forth between the two ends of the connection, but once the payload is delivered it just gets out of the way. TCP doesn't care what the payload is. It only guarantees that the message is sent and received appropriately.

When I open my message, I get another string of 1s and 0s, this time wrapped in the Hypertext Transfer Protocol (HTTP).

HTTP

```
telnet google.com 80
```

Trying 74.125.196.102...
Connected to google.com.
Escape character is '^]'.

HTTP is actually a text-based protocol, so it's human readable. If you'd like to see what it looks like, use the "telnet" command (which stands for "terminal over network"). The first argument is the server you'd like to connect to, the second is the port.

Most of the internet runs on port 80. That's the default. If you're making a secure connection (using the HTTPS or HTTP-Secure), the default port is 443.

After you run the telnet command, nothing much happens. The TCP connection is established, but you haven't sent any HTTP payload yet.

HTTP

GET

HTTP/1.0 200 OK
Date: Wed, 23 Apr 2014 16:01:22 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: PREF=ID=ee8f48ef18a74a2b:FF=0:TM=1398268882:LM=1398268882:S=_mP-DP8BMKSoUHPr; expires=Fri, 22-Apr-2016 16:01:22 GMT; path=/; domain=.google.com
Set-Cookie: NID=67=AQ4ubXjboMPdVqcU4puV9zUqPeRfqr6PjGgGV_4CL9Rv8vqd9WCbTUho5kTbCiuWizBM2AIBKkAxrREpwmDJ3N7-rVie0aaDciBbIUqY25_9UoSlV6Io5XNMlWSDpIIZ; expires=Thu, 23-Oct-2014 16:01:22 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

<!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">

Type the word GET, all-caps, and hit enter. That's when the magic happens.

GET is an HTTP method that instructs Google's server to send you something. Without an argument, Google by default sends you it's homepage. The response comes in three parts.

HTTP Status

HTTP/1.0 200 OK

The very first line gives you a summary of the response. The response is encoded in the HTTP version 1.0 protocol and the status is 200, which is code for “OK”, as in everything was “OK” with your request.

HTTP Response Headers

Date: Wed, 23 Apr 2014 16:01:22 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: PREF=ID=ee8f48ef18a74a2b:FF=0:TM=1398268882:LM=1398268882:S=_mP-DP8BMKSoUHPr; expires=Fri, 22-Apr-2016 16:01:22 GMT; path=/; domain=.google.com
Set-Cookie:
NID=67=AQ4ubXjboMPdVqcU4puV9zUqPeRfqr6PjGgGV_4CL9Rv8vqd9WCbTUho5kTbCiuWizBM2AIBKkAxrREpwmDJ3N7-rVie0aaDciBbIUqY25_9UoSlV6Io5XNMlWSDpIlZ; expires=Thu, 23-Oct-2014 16:01:22 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

The next section should look familiar. It's a Hash. The keys come before the colon, the values come after. This section contains a bunch of useful information *about* your response, but it's not the response yet.

HTTP Response Headers

```
Cache-Control: private, max-age=0
```

In the HTTP response headers you can see useful information about how the page should be cached, which we'll talk about later.

HTTP Response Headers

```
Set-Cookie: PREF=ID=ee8f48ef18a74a2b:FF=0:TM=1398268882:
LM=1398268882:S=_mP-DP8BMKSoUHPr; expires=Fri, 22-Apr-2016
16:01:22 GMT; path=/; domain=.google.com
Set-Cookie:
NID=67=AQ4ubXjboMPdVqcU4puV9zUqPeRfqr6PjGgGV_4CL9Rv8vqd9WC
bTUho5kTbCiuWizBM2AIBKkAxrREpwmDJ3N7-
rVie0aaDciBbIUqY25_9UoSlV6Io5XNMlWSDplIZ; expires=Thu, 23-
Oct-2014 16:01:22 GMT; path=/; domain=.google.com;
HttpOnly
```

You'll also see those "cookies" you've all probably heard about. Each "cookie" is itself a little Hash. Google is asking your browser to store these values, some for a year, others for only 6 months.

HTTP Response Headers

```
Content-Type: text/html; charset=ISO-8859-1
```

The Content-Type is also important. That tells you what the response is. Immediately following this response header section, you should expect a text/html document encoded with character set ISO-8859-1, which is an ASCII-based character encoding.

The Internet predates Unicode, so, at the HTTP level, computers still talk in ASCII.

HTTP Response Body

```
<!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">...
```

The rest of the response is the source code for Google's homepage. It's written in HTML, which we'll learn about later. This is what gets processed by your browser to display a web page. There's a lot here for such a simple webpage, way more than I can fit on this slide. But this whole block of text represents everything a web browser needs to display the Google homepage.

Behind the scenes, TCP chunked this entire response (header and body) into packets that got sent over the Internet, one at a time, and reconstituted into what you see here. Kind of like Star Trek teleportation.

HTTP Methods

GET

HEAD

POST

PUT

DELETE

TRACE, OPTIONS, CONNECT, PATCH

Recall that I used the word “GET” to get Google’s webpage. That’s just one of many HTTP methods. The most common are GET, HEAD, POST, PUT, and DELETE, which we’ll talk about. There are more obscure methods like TRACE, OPTIONS, CONNECT, and PATCH, which we won’t talk about.

HTTP GET

```
curl -i -X GET google.com

curl -i google.com
```

telnet is great for debugging at the TCP level, but if all we're interested in doing is making HTTP requests, we can use the curl command instead, the "client URL request library".

Like the man page says "curl offers a busload of useful tricks". The -i flag, for example, tells curl to output the response headers as well as the body. The -X flag allows us to specify the HTTP method. GET is the default method, so we don't have to specify it if we don't want to.

HTTP Status Codes

```
curl -i google.com
```

HTTP/1.1 301 Moved Permanently
Location: http://www.google.com/

```
curl -i www.google.com
```

HTTP/1.1 200 OK

You may have noticed that we didn't get a "200 OK" status code back this time. We got a "301 Moved Permanently". Numeric status codes are the quick-and-easy way a server can respond to an HTTP request. Everything else, including the response body, is optional. The browser can load a completely blank page, but behind the scenes there still must be a response code.

This particular response code, 301, says that Google's actual URL is www.google.com, not google.com. If we curl www.google.com, we'll get a 200.

HTTP Status Cats

http://httpcats.herokuapp.com/

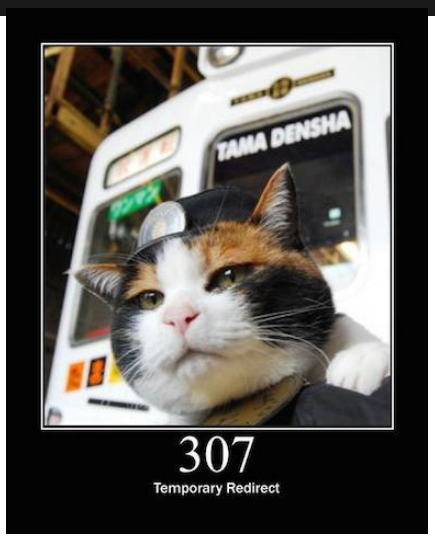
There are tons of HTTP status codes, some more obscure than others. Of course, since this is the internet, someone has associated cat photos to each of the codes.

404 Not Found

404, for example, is the status code you get when the server exists, but the page you're requesting doesn't. The page is "not found".

All the codes in the 400 block represent problems with the person making the request, known as the client. This could be a browser or even just the curl command itself.

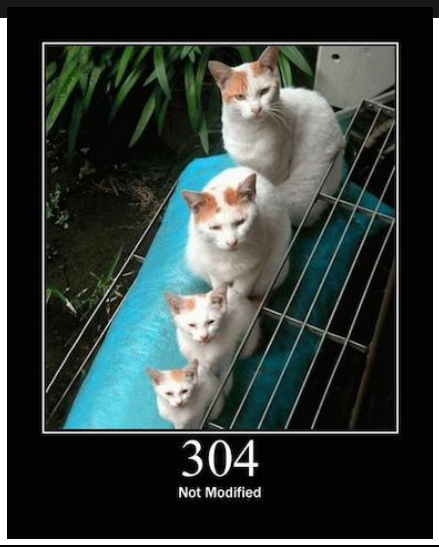
3XX Redirect



Most of the 300s status codes indicate that the page has moved. 301, which we've seen already, means that the page has been moved permanently, so you can update your bookmarks. 302 means the page has only been moved temporarily.

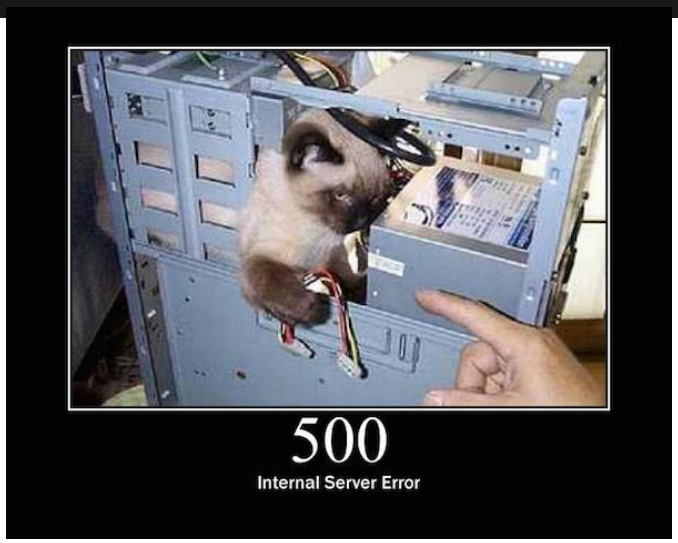
Due to some confusion on the part of the people who make browsers, 302 and 303 are similar to 307 and 301 is similar to 308. It's quite a mess, so most people just assume anything in the 300 block represents a redirect of some kind, even though that isn't actually true.

304 Not Modified



Case in point, 304. “Not Modified” is the kind of code you’d get when you refresh your browser. The server is saying “the stuff I sent you is still good, nothing new to report”, which the browser can use to just re-display the last page. This is called browser caching and it allows the page to load much faster.

5XX Server Error



When you write code that throws an error, the server will typically respond with a status code in the 500 range, usually 500 itself, which represents an “internal server error”. You’ll see this code a lot while you’re debugging your web application.

200 Ok

This is the code you want to see come back, the 200 OK. Anything in the 200 range is good. 201 means that you tried to create an object in the server, and you were successful. 204 is like 200, but it lets you know the page will be blank on purpose.

HTTP HEAD

```
curl -i -X HEAD www.google.com

curl -iX HEAD www.google.com

curl -X HEAD -i www.google.com
```

Enough about status codes. You can look them up yourself online when you see one you don't remember. That's what I do.

The HTTP HEAD method is like the GET method, except without the content. It's a way for you to sample what the content might be by peeking at the response headers. You can then later decide if you'd like to GET the whole webpage as well. It's a quick way to access a site if you're just interested in checking the status code.

A note about the curl command: You can combine the i and X flags to curl, but they have to be in the right order. The request method has to come after the X flag. If you want to put the X flag first, you have to split the flags.

HTTP POST/PUT

```
curl -iX POST --data "" www.google.com

curl -iX PUT --data "" www.google.com
```

The POST and PUT methods are similar. They are used when the client (e.g. a browser) wants to send data to a server. They're used in online forms, for example.

POST is supposed to be used for creating objects in the server. PUT should be used for updating existing objects.

When used via curl, you need to pass some data with these methods. Otherwise Google responds with a status 411 Length Required, which is a little confusing. When you pass no data, it has no length. And if you have no length when it's required (such as during a POST or PUT), then you get a 411.

With the data flag set, even an empty one, curl will tell Google that the length of your data is 0 rather than nil. In that case you get the much clearer 405 Method Not Allowed status, which is self-explanatory.

HTTP DELETE

```
curl -iX DELETE www.google.com
```

The DELETE method is also pretty obvious. You use this method when you want to delete some object in the server.

As with POST and PUT, Google responds with a 405 Not Allowed because this HTTP method is not allowed on www.google.com.

HTTP Methods

All the method descriptions I've listed so far are only conventions. Actually I'd say they're more than conventions. They're documented in the HTTP Specification this way. But it's up to the server programmers (people like you) to implement how the server responds to these methods. And programmers are rule breakers. So you'll commonly see weird HTTP methods and bad status codes out "in the wild". Typically bad sites will treat *everything* as a GET and only respond with either 200 OK, 500 Error, or 404 Not Found. They'll ignore everything else.

REST

http://yourdomain.com/things

GET = List all the things
PUT = Replace all the things
POST = Create a new thing
DELETE = Delete all the things

One the opposite end of the spectrum are the folks who follow the HTTP Spec to the letter and then some. These servers are said to be RESTful, meaning that they implement an HTTP REST architecture. REST stands for “representational state transfer” and was introduced in 2000 by UC Irvine computer scientist Roy Fielding.

In a RESTful server, the URLs follow a conventional pattern of your domain followed by a plural noun. Doing a GET to this URL should return a list of all the things. A PUT should replace all the things with the data in the request. A POST should create a single new thing and add it to the list of things. And a DELETE should delete all the things.

REST

http://yourdomain.com/things/thing17

GET: Get the 17th thing.
PUT: Update the 17th thing.
POST: Create the 17th thing.
DELETE: Delete the 17th thing.

When operating on a single item in a collection, rather than the entire collection, the URL pattern is the plural noun, followed by an identifier.

When issuing a GET to this URL, you should get a page about the specified thing - in this case it's the 17th thing in the collection of things.
When you PUT some data, the server should use that data to update the thing.
When you POST data, the server should create a 17th thing.
And when send a DELETE request, the server should delete the 17th thing.

Rails, which we'll learn about later, follows this pattern. You'll find it very easy to create a RESTful server with Rails.

URL

scheme://domain:port/path?query_string#fragment_id

http://ip-address:80/path?query_string#fragment_id

So far we've described almost the entire web address, also known as a URL or "uniform resource locator".

The scheme part of the URL says what kind of protocol I'd like to use in my connection. In most cases it's http or https.
The domain is turned into an ip-address by DNS and is used by the IP to locate another computer on the network.
The port is 80 by default (443 for HTTPS), and tells where I should start my connection.
The path describes the resource I'd like to access. Usually this is a file. Sometimes it's a RESTful path.

The two parts we haven't talked about, the query_string and fragment_id, are used to pass additional information along with the request. Much like a Hash, it's usually a collection of keys and values. We'll see more of these when we start building websites.

Cookies

One last thing I want to talk about is cookies. Like we saw earlier with the request to www.google.com, Google sends a response containing cookies.

Imagine a cookie like it's a sticker. Imagine that every time you talk into a store, the shopkeeper puts a sticker on your shirt. The sticker contains some information, like an id number or your name. It also has an expiration date, after which the sticker falls off.

You're probably thinking - wow, that's really annoying. If I take one trip to the mall, I'll come home with stickers all over myself. And, when you put it like that, it *is* annoying. But HTTP cookie stickers as actually really important.

Cookies

Set-Cookie: PREF=ID=4f6a4c062bfcdd20:FF=0:TM=1398403479:LM=1398403479:S=BDWwfz1Y6jrs63aC; expires=Sun, 24-Apr-2016 05:24:39 GMT; path=/; domain=.google.com

HTTP is a stateless protocol. That means that every request to the server is treated like a brand new person. It doesn't matter that the request is originating from the same IP address, or that it's the 2nd/3rd/4th/5th request in a row from the same computer. As far as HTTP is concerned, every request is new and unique.

That's a problem. If every request is new, then there would be no way to login to a website. After entering my username and password, the server will immediately forget who I am on the very next page.

Cookies fix that problem. The server, through a Set-Cookie response header like this one, asks your browser to keep track of some data. The browser then promises to send that same data back to the server on every request. That way, the server can use that cookie to figure out that a second and third request is coming from the same person as the first request. And that's how you keep track of a logged-in session.

Cookies

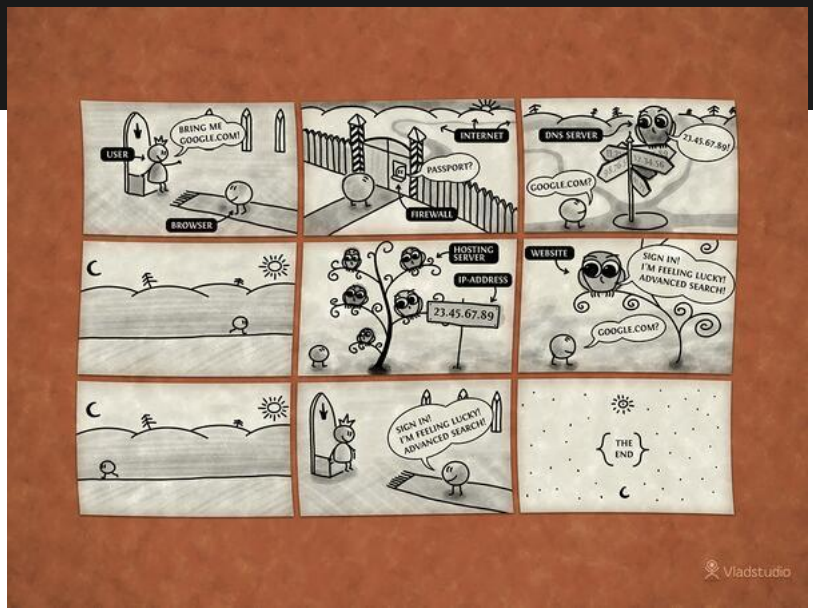
While cookies can be used for good by allowing you to create private places on the web that only you can access, be it your Facebook photos or your bank account, they can also be used for evil.

Ads, like this one here which you might have seen, also use cookies. When you visited wyncode.co, we asked your browser to store a cookie for us. That cookie is used to identify you on sites other than wyncode.co. Those sites will display ads like this one, reminding you of that awesome code school site you visited, over and over again.

You're free to delete your cookies if you'd like. That option is hiding in your browser settings somewhere. But by deleting all your cookies, you'll also log yourself out of everything - your Facebook page, your Twitter account, everything. They'll all forget who you are and ask you to re-identify yourself by logging in again.

Cookies

So we just have to live with cookies. We just have to live with strangers putting stickers all over us.



In a nutshell.