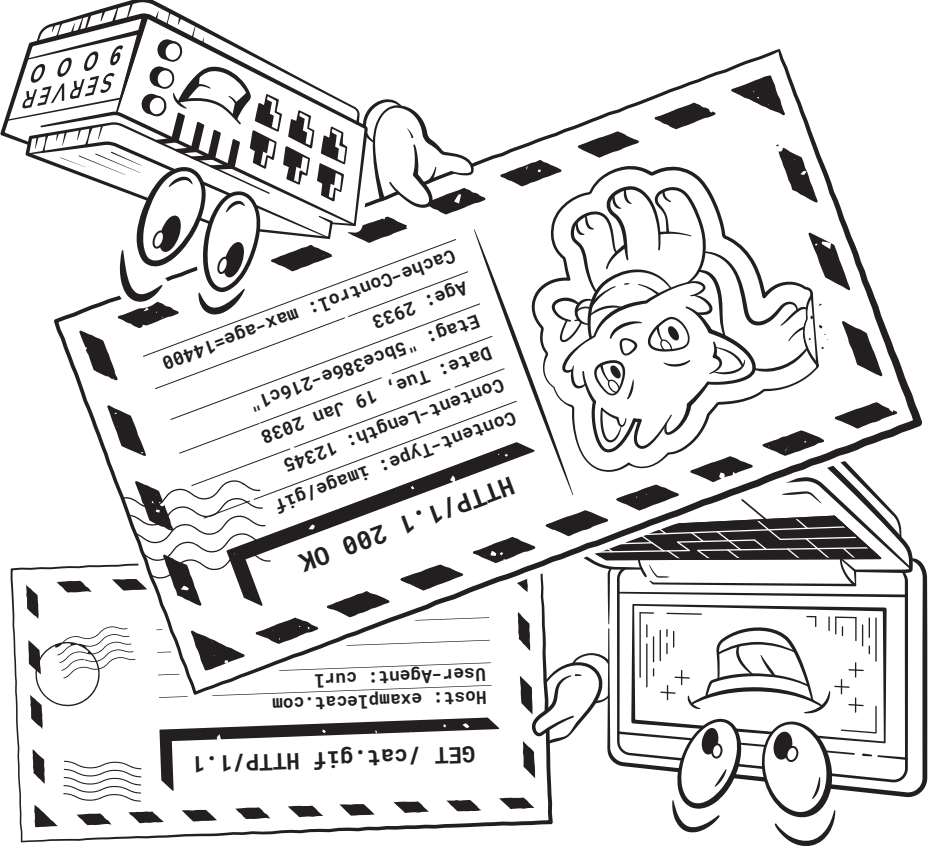


# HTTP

Learn-your  
browsers language



by Julia Evans

like this?  
more zines at  
wizardsines.com

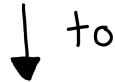
# about this zine

Your browser uses HTTP every time it visits a website. Like a lot of the tech that runs the internet, understanding HTTP isn't that hard!

This zine's goal is to take you from:



my website isn't working right because of some weird caching/cookies/CORS thing and I'm not sure where to start



hmm, I have a caching problem, I can just look at my request+response headers, consult the documentation, and figure out how to fix it!

published by wizard industries (<https://wizardzines.com>)  
cover art by Vladimir Kašiković

# how to learn more

## ♥ Mozilla Developer Network

<https://developer.mozilla.org>

MDN is a fantastic wiki maintained by Mozilla. It has tutorials and reference documentation for HTML, CSS, HTTP, Javascript. It's the best place to start for reference documentation.

## ♥ OWASP

<https://cheatsheetseries.owasp.org>

OWASP is an organization that publishes security best practices. If you have a question about web security, they've probably published a cheat sheet or guide to help you.

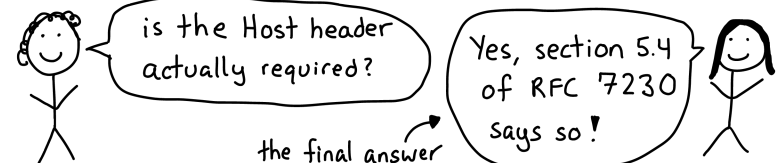
## ♥ httpstatuses.com

Nice little site that explains all the HTTP status codes.

## ♥ RFCs

<https://tools.ietf.org/html/rfcXXXX> <sup>put RFC number here</sup>

RFCs are numbered documents (like "RFC 2631"). Every Internet protocol (like TLS or HTTP) has an RFC. These are where you go to find the Official Final Answers to technical questions you have about any internet standard. The HTTP standard is mostly documented in 6 RFCs numbered 7230 to 7235.



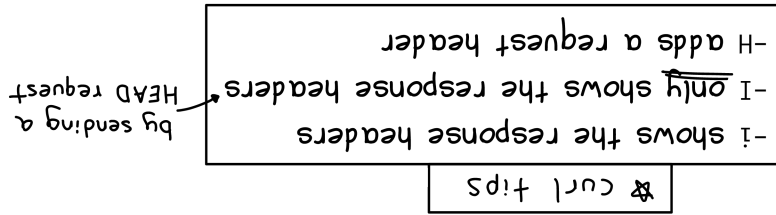
Don't be scared of using an RFC if you want to know for sure!

# Table of Contents

- 4 ..... what's HTTP?
- 5 ..... how URLs work
- 6 ..... what's a header?
- 7 ..... anatomy of an HTTP request
- 8-9 ..... request methods (GET & POST)
- 10 ..... request headers
- 11 ..... using HTTP APIs
- Responses:
- 12 ..... anatomy of an HTTP response
- 13 ..... response headers
- 14 ..... status codes (200 & 404)
- 15 ..... how cookies work
- 16-17 ..... content delivery networks & caching
- 18 ..... redirects
- 19 ..... HTTP/2
- security:
- 20-21 ..... HTTPS & certificates
- 22-24 ..... same origin policy & CORS
- 25 ..... security headers
- 26-27 ..... exercises & how to learn more

# HTTP exercises

Making HTTP requests with curl to real internet websites and trying different headers is my favourite way to play around with HTTP & learn.



Try the Range header:

```
curl -i https://examplecat.com/cat.txt -H "Range: bytes=8-17"
```

Request (and print out!) a compressed response:

```
curl -i https://examplecat.com -H "Accept-Encoding: gzip" -- output -
```

Get a webpage in Spanish:

```
curl -i https://twitter.com -H "Accept-Language: es-ES"
```

Get redirected to another URL:

```
curl -i http://examplecat.com
```

(hint: look at the Location header!)

Guess what content delivery network Github is using:

```
curl -I https://github.githubassets.com
```

(hint: it's in a header starting with X-)

Find out when example.com was last updated

```
curl -I example.com
```

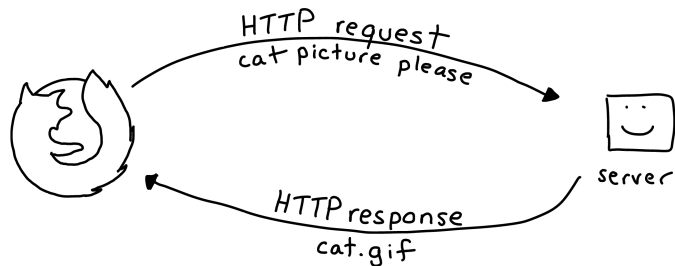
(hint: Last-Modified)

Get a 404 not found

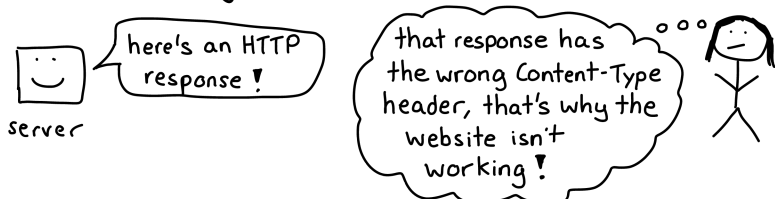
```
curl -i examplecat.com/bananas
```

# what's HTTP?

HTTP is the protocol (Hypertext Transfer Protocol) that's used when you visit any website in your browser.



The exciting thing about HTTP is that even though it's used for literally every website, HTTP requests and responses are easy to look at and understand:



Example of what an HTTP request and response might look like:

	request		response
request line	{ GET / HTTP/1.1	status	{ HTTP/1.1 200 OK
headers	{ Host: examplecat.com User-Agent: curl Accept: */*	headers	{ Cache-Control: max-age=604800 Content-Type: text/html Etag: "1541025663+ident" Server: ECS (nyb/1D0B) Vary: Accept-Encoding X-Cache: HIT Content-Length: 1270
		body	{ <!doctype html> <title>Example Cat</title> ...

All that text is a lot to understand, so let's get started learning what all of it means!

# security headers

These are headers your server can set. They ask the browser to protect your users' data against attackers in different ways:

**Content-Security-Policy** often called CSP

Only allow CSS / Javascript from certain domains you choose to run on your website. Helps protect against cross-site-scripting (aka XSS) attacks.

**Referrer-Policy**

Control how much information is sent to other sites in the Referrer header. Example: Referrer-Policy: no-referrer. spelling is inconsistent with Referrer header !!

**Strict-Transport-Security** often called HSTS

Require HTTPS. If you set this the client (browser) will never request a plain HTTP version of your site again. Be careful! You can't take it back!

**Expect-CT**

Certificate Transparency (CT) is a system that can help find malicious SSL certificates issued for your site. This header gives the browser a URL to use to report bad certificates to you.

**X-XSS-Protection**

Another way to protect against XSS attacks. Not supported by all browsers, Content-Security-Policy is more powerful.

# how URLs work

https://examplecat.com:443/cats?color=light%20gray#banana  
 scheme domain port path query string fragment id

**scheme** Protocol to use for the request. Encrypted (https), insecure (http), or something else entirely (ftp).  
**domain** Where to send the request. For HTTP(s) requests, the host header gets set to this (Host: example.com)

**port** Defaults to 80 for HTTP and 443 for HTTPS. :443

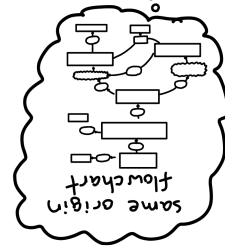
**path** Path to ask the server for. The path and the query parameters are combined in the request, like: GET /cats?color=light%20gray HTTP/1.1

**query parameters** Query parameters are usually used to ask for a different version of a page ("I want a light gray cat"). Example:  
 hair=short&color=black&name=mr%20darcy  
 name=value separated by &

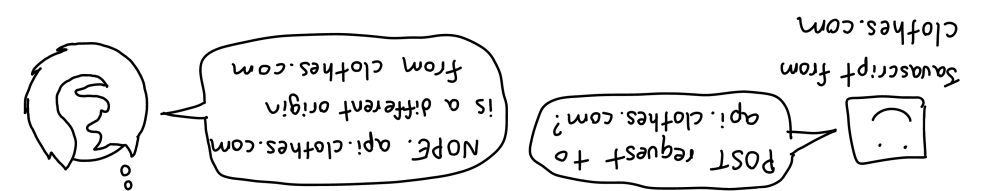
**URL encoding** URLs aren't allowed to have certain special characters like spaces, @, etc. So to put them in a URL you need to percent encode them as % + hex representation of ASCII value.  
 space is %20, % is %25, etc.

**fragment id** This isn't sent to the server at all. It's used either to jump to an HTML tag (<a id="banana">...) or by javascript on the page.  
 #banana

# CORS

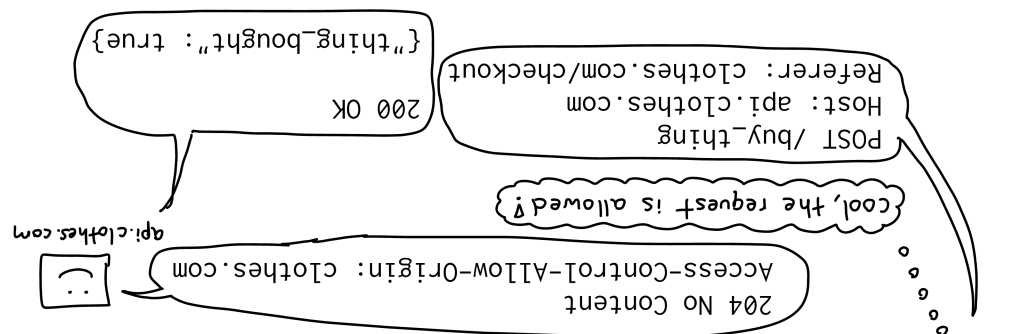
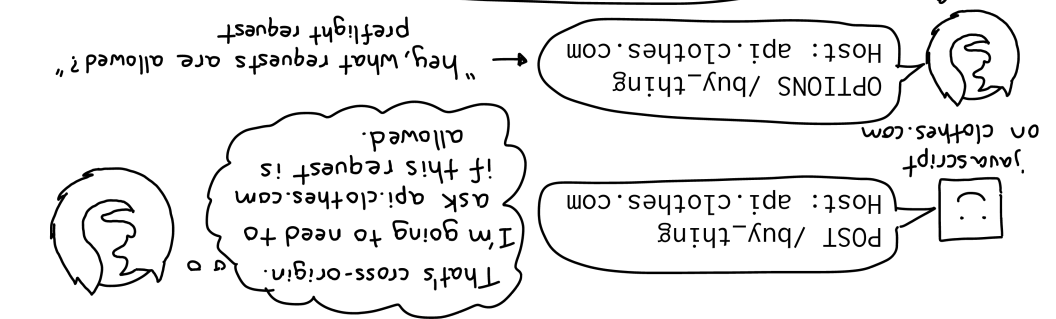


Cross-origin requests are not allowed by default: (because of the same origin policy!)



If you run api.clothes.com, you can allow clothes.com to make requests to it using the Access-Control-Allow-Origin header.

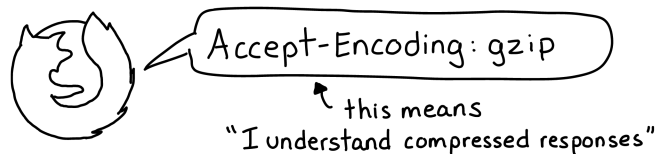
Here's what happens:



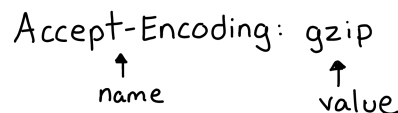
This OPTIONS request is called a "preflight" request, and it only happens for some requests, like we described in the diagram on the same-origin policy page. Most GET requests will just be sent by the browser without a preflight request, but POST requests that send JSON need a preflight.

# what's a header?

Every HTTP request and response has headers. Headers are a way for the browser or server to send extra information!



Headers have a name and a value.



Header names aren't case sensitive:

totally valid → aCcEpT-eNcOdIng: gzip

There are a few different kinds of headers:

## Describe the body:

Content-Type: image/png    Content-Encoding: gzip  
Content-Length: 12345    Content-Language: es-ES

## Ask for a specific kind of response:

Accept: image/png            Accept-Encoding: gzip  
Range: bytes=1-10            Accept-Language: es-ES

## Manage caches:

ETag: "abc123"                    If-Modified-Since: 3 Aug 2019 13:00:00 GMT  
If-None-Match: "abc123"        Last-Modified: 3 Feb 2018 11:00:00 GMT  
Vary: Accept-Encoding           Expires: 27 Sep 2019 13:07:49 GMT  
Cache-Control: public, max-age=300

## Say where the request comes from:

User-Agent: curl                    Referer: https://examplecat.com

## Cookies:

Set-Cookie: name=julia; HttpOnly (server → client)  
Cookie: name=julia (client → server)

and more!

# why the same origin policy matters

Browsers work hard to make sure that evil.com can't make requests to other-website.com. But evil.com can request other-website.com from its own server, what's the big deal?

2 reasons it's important to restrict Javascript on websites from making arbitrary requests from your browser:

## Reason 1: cookies

Browsers often send your cookies with HTTP requests. You don't want evil.com to be able to make requests using your login cookies. They'd be logged in as you!



## Reason 2: network access

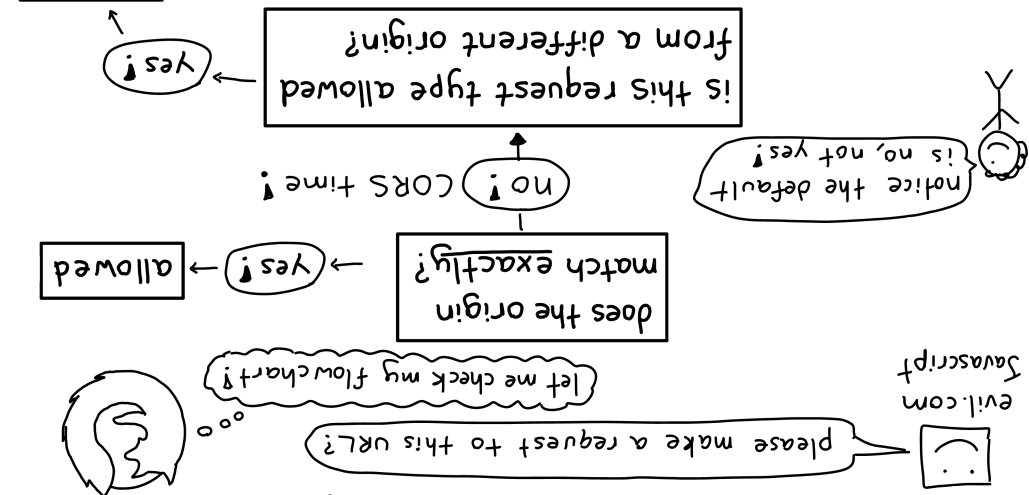
You might be on a private network (for example your company's corporate network) that evil.com doesn't have access to, but your computer does.



# the same origin policy

An origin is the protocol + domain including subdomains + port  
 example: `https://fabby.examplecat.com:443`

The same origin policy is one way browsers protect you from malicious javascript code. Here's basically how it works:



evil.com javascript  
 please make a request to this URL?  
 let me check my flowchart!

# anatomy of an \* HTTP request \*

HTTP requests always have:

→ a domain (like `examplecat.com`)

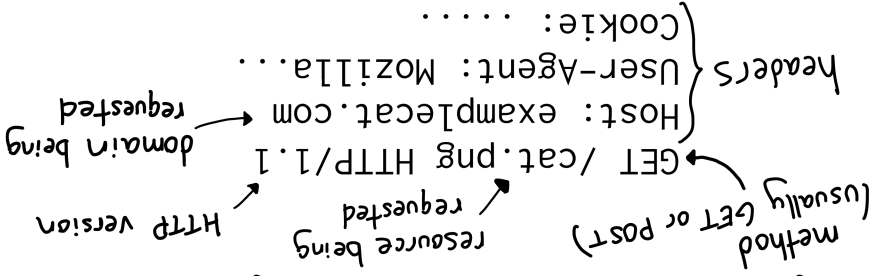
→ a resource (like `/cat.png`)

→ a method (GET, POST, or something else)

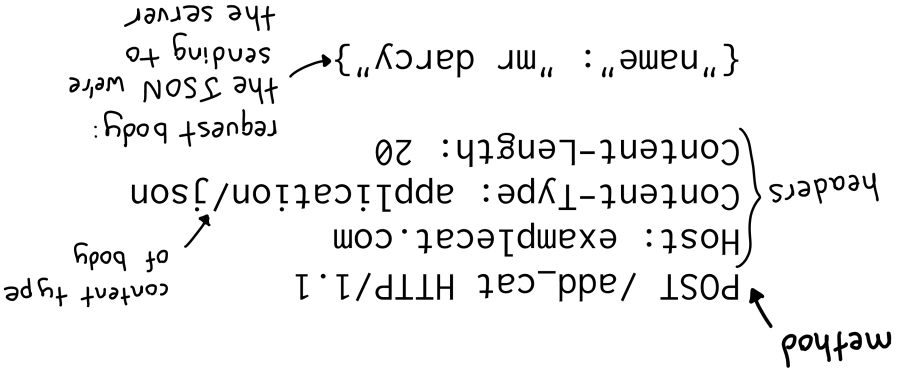
→ headers (extra information for the server)

There's an optional request body. GET requests usually don't have a body, and POST requests usually do.

This is an HTTP 1.1 request for `examplecat.com/cat.png`. It's a GET request, which is what happens when you type a URL in your browser. It doesn't have a body.



Here's an example POST request with a JSON body:



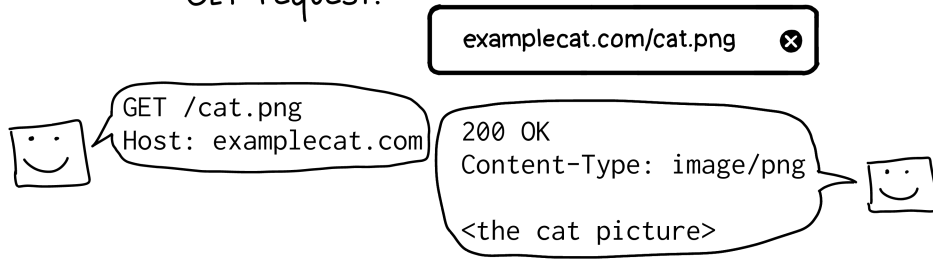
# request methods

Every HTTP request has a method. It's the first thing in the first line:

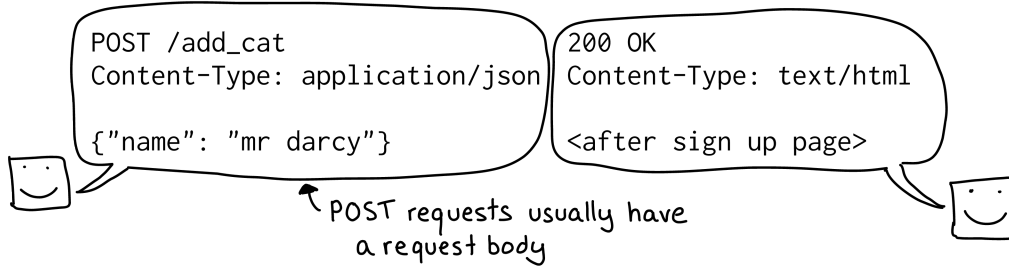
↙ this means it's a GET request  
GET /cat.png HTTP/1.1

There are 9 methods in the HTTP standard. 80% of the time you'll only use 2 (GET and POST).

**GET** When you type an URL into your browser, that's a GET request.



**POST** When you hit submit on a form, that's (usually) a POST request.



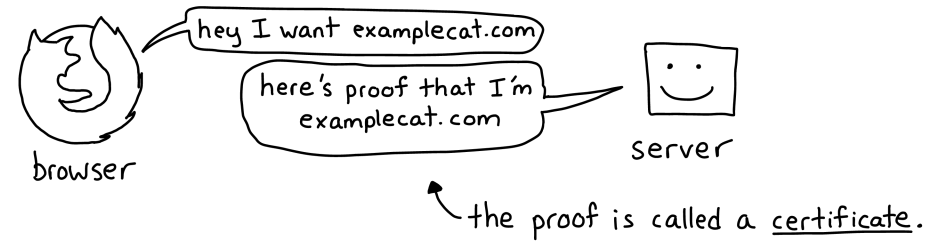
The big difference between GET and POST is that GETs are never supposed to change anything on the server.

**HEAD** Returns the same result as GET, but without the response body.



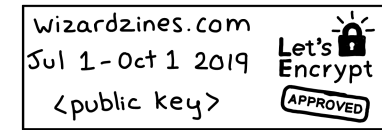
# certificates

To establish an HTTPS connection to `examplecat.com`, the client needs proof that the server actually is `examplecat.com`.



A TLS certificate has:

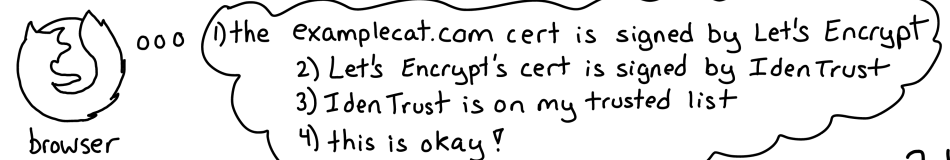
- a set of domains it's valid for (eg `examplecat.com`)
  - a start and end date (example: july 1 2019 to oct 1 2019)
  - a secret private key which only the server has
  - a public key to use when encrypting
  - a cryptographic signature from someone trusted
- this is the only secret part, the rest is public



The trusted entity that signs the certificate is called a **★ Certificate Authority ★ (CA)** and they're responsible for only signing certificates for a domain for that domain's owner.



When your browser connects to `examplecat.com`, it validates the certificates using a list of trusted CAs installed on your computer. These CAs are called "root certificate authorities".



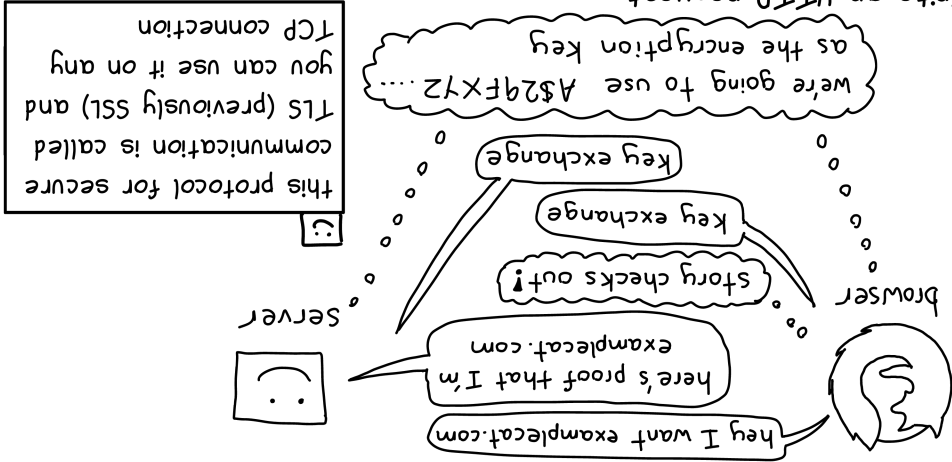


# HTTPS: HTTP + secure

Here's what your browser does when it asks for `https://examplecat.com/cat.png`

① Negotiate an encryption key (AES symmetric key) to use for this connection to `examplecat.com`. The browser and server will use the same key to encrypt/decrypt content.

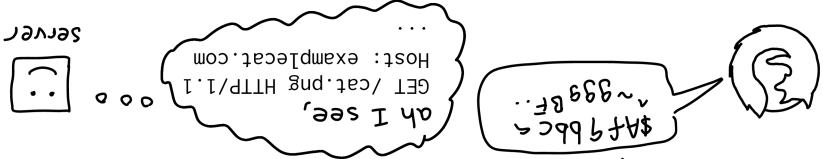
simplified version of how picking the encryption key works:



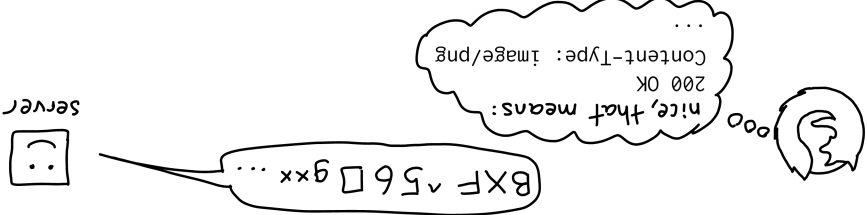
② Write an HTTP request

```
GET /cat.png HTTP/1.1
Host: examplecat.com
User-Agent: Mozilla/...
```

③ Encrypt the HTTP request with AES & send it to `examplecat.com`



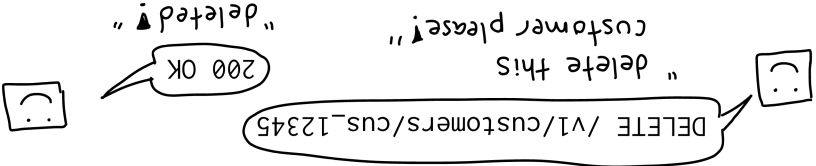
④ Receive encrypted HTTP response



## OPTIONS

OPTIONS is mostly used for CORS requests. The CORS page has more about that. It also tells you which methods are available.

DELETE Used in many APIs (like the Stripe API) to delete resources.



## PUT

Used in some APIs (like the S3 API) to create or update resources. PUT /cat/1234 lets you GET /cat/1234 later.

## PATCH

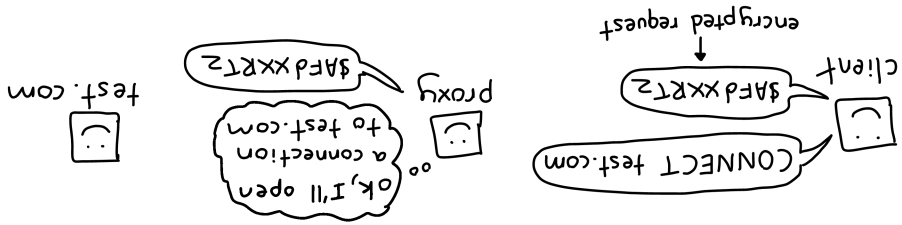
Used in some APIs for partial updates to a resource ("just change this 1 field").

## TRACE

I've never seen a server that supports this, you probably don't need to know about it.

## CONNECT

Different from all the others: instead of making a request to a server directly, it asks for a proxy to open a connection. If you set the `HTTPS_PROXY` environment variable to a proxy server, many HTTP libraries will use this protocol to proxy your requests.



# request headers

These are the most important request headers:

## Host

The domain.  
The only required header.

Host: examplecat.com

## User-Agent

name + version  
of your browser  
and OS

User-Agent: curl 7.0.2

## Referer

website that linked  
or included the resource

Referer: https://examplecat.com  
*yes, it's misspelled!*

## Authorization

eg a password  
or API token

base64 encoded user: password  
Authorization: Basic YXZ

## Cookie

send cookies the  
server sent earlier  
keeps you logged in.

Cookie: user=b0rk

## Range

lets you continue  
downloads ("get  
bytes 100-200")

Range: bytes=100-200

## Cache-Control

"max-age = 60"  
means cached  
responses must be  
less than 60  
seconds old

## If-Modified-Since

only send if resource  
was modified after  
this time

If-Modified-Since: Wed, 21 Oct...

## If-None-Match

only send if the  
ETag doesn't match  
those listed

If-None-Match: "e7ddac"

## Accept

MIME type you  
want the response  
to be

Accept: image/png

## Accept-Encoding

set this to "gzip"  
and you'll probably get  
a compressed response

Accept-Encoding: gzip

## Accept-Language

set this to "fr-CA"  
and you might get a  
response in French

Accept-Language: fr-CA

## Content-Type

MIME type of  
request body, e.g.  
"application/json"

## Content-Encoding

will be "gzip" if  
the request body  
is gzipped

## Connection

"close" or "keep-alive".  
Whether to keep the  
TCP connection open.

# HTTP/2

HTTP/2 is a new version of HTTP.

Here's what you need to know:

## ★ A lot isn't changing

All the methods, status codes, request/response bodies,  
and headers mean exactly the same thing in HTTP/2.

before (HTTP/1.1)	one change:	after (HTTP/2)
method: GET	Host header	method: GET
path: /cat.gif	=> authority	path: /cat.gif
headers:		authority: examplecat.com
- Host: examplecat.com		headers:
- User-Agent: curl		- User-Agent: curl

## ★ HTTP/2 is faster

Even though the data sent is the same, the way HTTP/2 sends it  
is different. The main differences are:

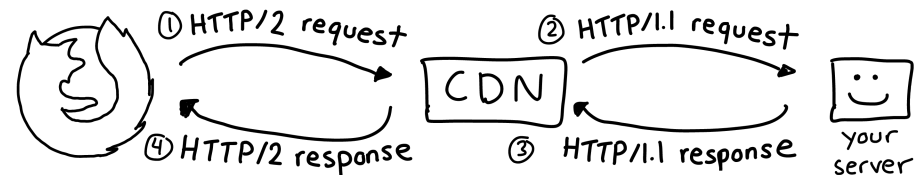
- It's a binary format (it's harder to tcpdump traffic and debug)
- Headers are compressed
- Multiple requests can be sent on the same connection at a time

before (HTTP/1.1)	after (HTTP/2)	} one TCP connection
→ request 1 response 1 ← → request 2 response 2 ←	→ request 1 → request 2 out of order is OK { response 2 ← response 1 ←	

All these changes together mean that HTTP/2 requests often  
take less time than the same HTTP/1.1 requests.

## ★ Sometimes you can switch to it easily

A lot of software (CDNs, nginx) let clients connect with  
HTTP/2 even if your server still only supports HTTP/1.1.



# using HTTP APIs

lots of services (Twitter! Twilio! Google!) let you use them by sending them HTTP requests. If an HTTP API doesn't come with a client library, don't be scared! You can just make the HTTP requests yourself. Here's what you need to remember:

→ Set the right **Content-Type** header

Often you'll be sending a POST request with a body, and that means you need a Content-Type header that matches the body.

The 2 main options are:

\* application/json → JSON!

\* application/x-www-form-urlencoded  
← same as what a HTML form does

If you don't set the Content-Type, your request won't work.

a common error is to try to send POST data as one content type (like JSON) when it's actually another (like application/x-www-form-urlencoded)



→ Identify yourself

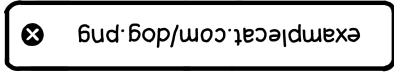
Most HTTP APIs require a secret API key so they know who you are. Here's how that looks for the Twilio API:

```
curl https://api.twilio.com/2010-04-01/Accounts/ACCOUNT_ID/Messages.json
-H "Content-Type: application/json"
-u ACCOUNT_ID:AUTH_TOKEN
-d '{
  "from": "+15141234567",
  "to": "+15141234567",
  "body": "a text message"
}'
# this sends a POST request
```

-u sends the username/password in the Authorization header

# redirects

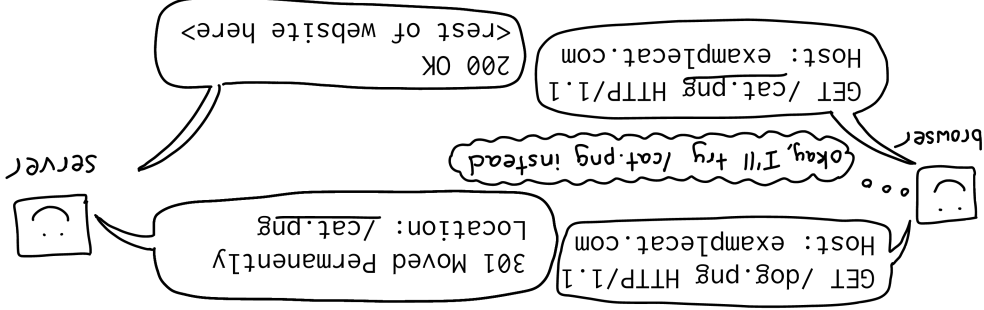
Sometimes you type a URL into your browser:



but end up at a slightly different URL: oh, where did the cat come from? I didn't type that!



Here's what's going on behind the scenes:



The **Location** header tells the browser what new URL to use. The new URL doesn't have to be on the same domain: examplecat.com/panda can redirect to pandas.com.

Setting up redirects is a great thing to do if you move your site to a new domain!

## Warning!

301 Moved Permanently redirects are PERMANENT: after a browser sees one once, it'll always use examplecat.com/cat.png when someone types examplecat.com/dog.png forever. You can't take it back and decide to not to redirect. If you're not sure you want to redirect your site for eternity, use 302 found to redirect instead.

# anatomy of an HTTP response

HTTP responses have:

- a status code (200 OK! 404 not found!)
- headers
- a body (HTML, an image, JSON, etc)

Here's the HTTP response from `examplecat.com/cat.txt`:

```

HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Content-Length: 33
Content-Type: text/plain; charset=UTF-8
Date: Mon, 09 Sep 2019 01:57:35 GMT
Etag: "ac5affa59f554a1440043537ae973790-ssl"
Strict-Transport-Security: max-age=31536000
Age: 0
Server: Netlify
    
```

Annotations:   
 - `200 OK` is labeled as **status** and **status code**.   
 - Headers are grouped by a bracket labeled **headers**.   
 - The body `\ / \ ( ' ) ( / ) \ ( _ ) |` is labeled as **body** and **cat! ☺**.

There are a few kinds of response headers: when the resource was sent/modified:

```

Date: Mon, 09 Sep 2019 01:57:35 GMT
Last-Modified: 3 Feb 2017 13:00:00 GMT
    
```

about the response body:

```

Content-Language: en-US
Content-Type: text/plain; charset=UTF-8
Content-Length: 33
Content-Encoding: gzip
    
```

caching:

```

Etag: "ac5affa..."
Age: 255
Vary: Accept-Encoding
Cache-Control: public, max-age=0
    
```

security: (see page 25)

```

X-Frame-Options: DENY
X-XSS-Protection: 1
Strict-Transport-Security: max-age=31536000
Content-Security-Policy: default-src https:
    
```

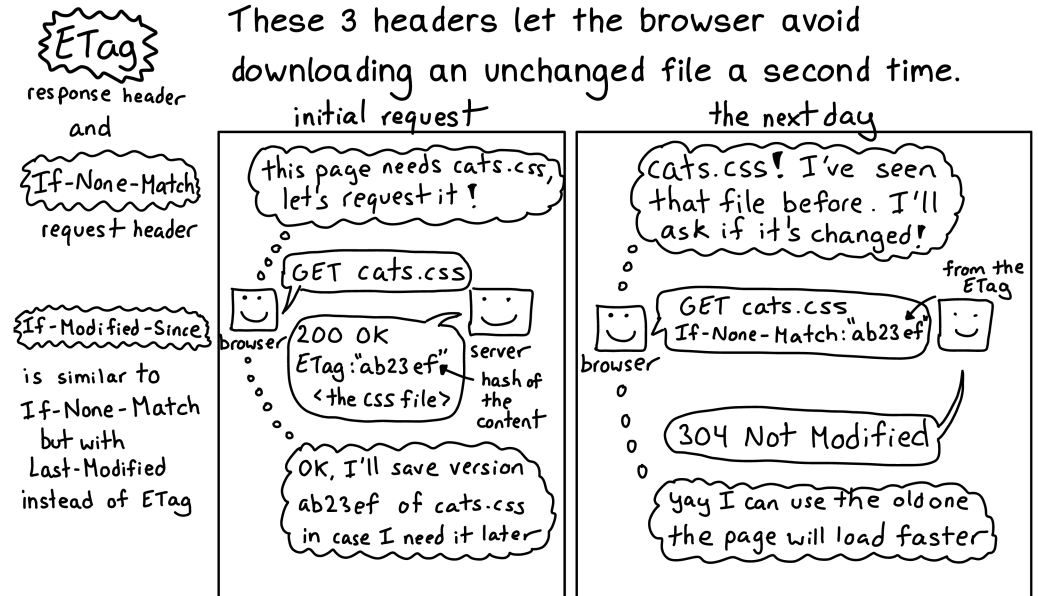
and more:

```

Connection: keep-alive
Via: nginx
Set-Cookie: cat=darcy; HttpOnly; expires=27-Feb-2020 13:18:57 GMT;
    
```

# caching headers

These 3 headers let the browser avoid downloading an unchanged file a second time.



**Vary** response header

Sometimes the same URL can have multiple versions (spanish, compressed or not, etc). Caches categorize the versions by request header like this:

Accept-Language	Accept-Encoding	content
en-US	-	hello
es-ES	-	hola
en-US	gzip	f\$xx99aef^.. (compressed gibberish)

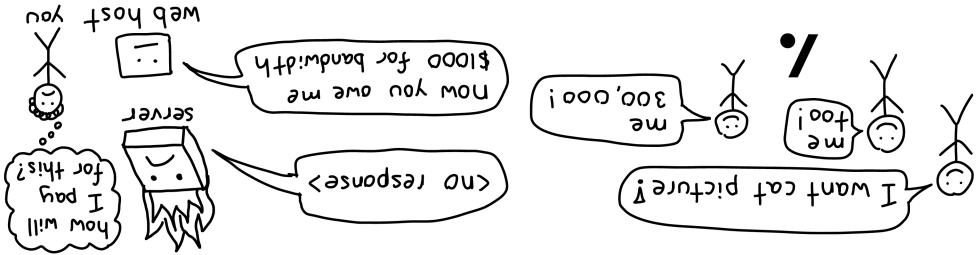
The Vary header tells the cache which request headers should be the columns of this table.

**Cache-Control** request AND response header

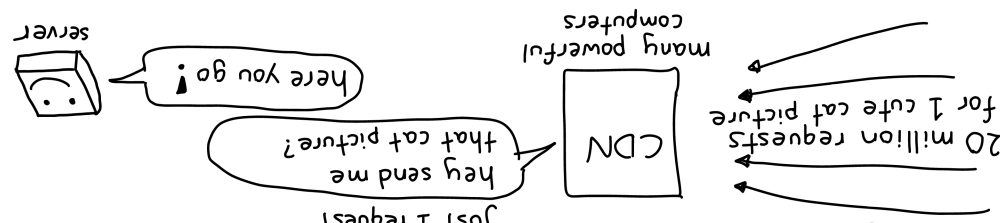
Used by both clients and servers to control caching behaviour. For example: `Cache-Control: max-age=9999999999` from the server asks the CDN or browser to cache the thing for a long time.

# content delivery networks

In 2004, if your website suddenly got popular, often the webserver wouldn't be able to handle all the requests.

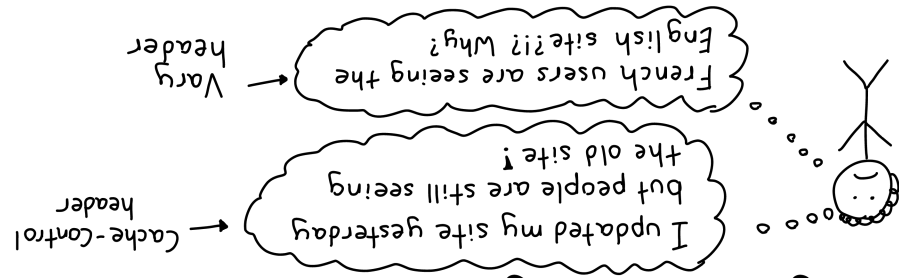


A CDN (content delivery network) can make your site faster and save you money by caching your site and handling most requests itself.



Today, there are many free or cheap CDN services available, which means if your site gets popular suddenly you can easily keep it running!

Next, we'll explain the HTTP headers your CDN or browser uses to decide how to do caching.



# response headers

**Age**

how many seconds response has been cached  
Age: 355

**Etag**

Version of response body  
Etag: "ac5afafa.."

**Via**

added by proxy servers  
Via: nginx

**Set-Cookie**

Sets a cookie.  
Set-Cookie: name=value; HttpOnly

**Content-Type**

MIME type of body  
Content-Type: text/plain

**Content-Language**

Language of body  
Content-Language: en-US

**Location**

URL to redirect to  
Location: /cat.png

**Date**

when response was sent  
Date: Mon, 09 Sep 2019...

**Cache-Control**

various caching settings  
Cache-Control: max-age=300

**Expires**

The response is stale and should be re-requested after this time.

**Vary**

request headers that response will vary based on

**Connection**

"close" or "keep-alive" whether to keep the TCP connection open

**Access-Control-\***

Called CORS headers. These allow cross-origin requests.

**Content-Length**

length of body in bytes  
Content-Length: 33

**Content-Encoding**

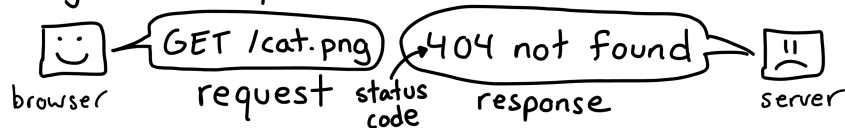
whether body is compressed  
Content-Encoding: gzip

**Accept-Ranges**

Whether Range request header is supported for this resource

# HTTP status codes

Every HTTP response has a **★ status code ★**.



There are 50ish status codes but these are the most common ones in real life:

200 OK

} 2xxs mean **★ success ★**

301 Moved Permanently

302 Found

temporary redirect

304 Not Modified

the client already has the latest version, "redirect" to that

} 3xxs aren't errors, just redirects to somewhere else

400 Bad Request

403 Forbidden

API key/OAuth/something needed

404 Not Found

we all know this one :)

429 Too Many Requests

you're being rate limited

} 4xx errors are generally the client's fault: it made some kind of invalid request

500 Internal Server Error

the server code has an error

503 Service Unavailable

could mean nginx (or whatever proxy)

couldn't connect to the server

504 Gateway Timeout

the server was too slow to respond

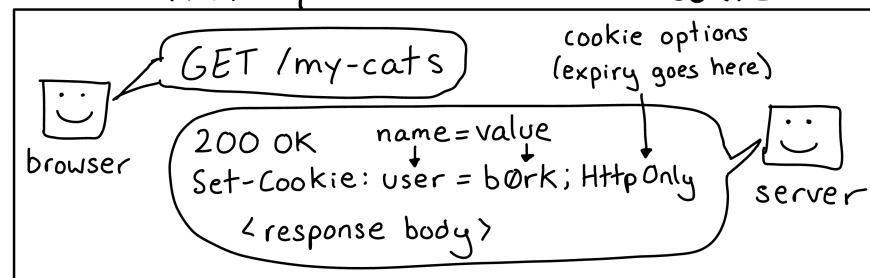
} 5xx errors generally mean something's wrong with the server.

# how cookies work

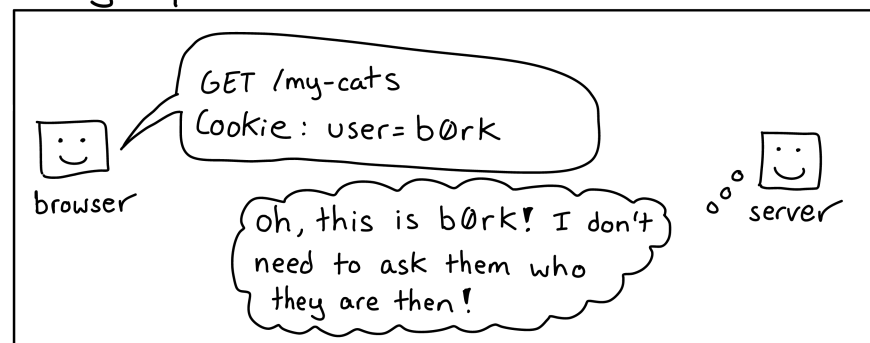
Cookies are a way for a server to store a little bit of information in your browser.

They're set with the Set-Cookie response header, like this:

first request: server sets a cookie



Every request after: browser sends the cookie back



Cookies are used by many websites to keep you logged in. Instead of user=b0rk they'll set a cookie like sessionid=long-incomprehensible-id. This is important because if they just set a simple cookie like user=b0rk, anyone could pretend to be b0rk by setting that cookie!

Designing a secure login system with cookies is quite difficult — to learn more about it, google "OWASP Session Management Cheat Sheet".