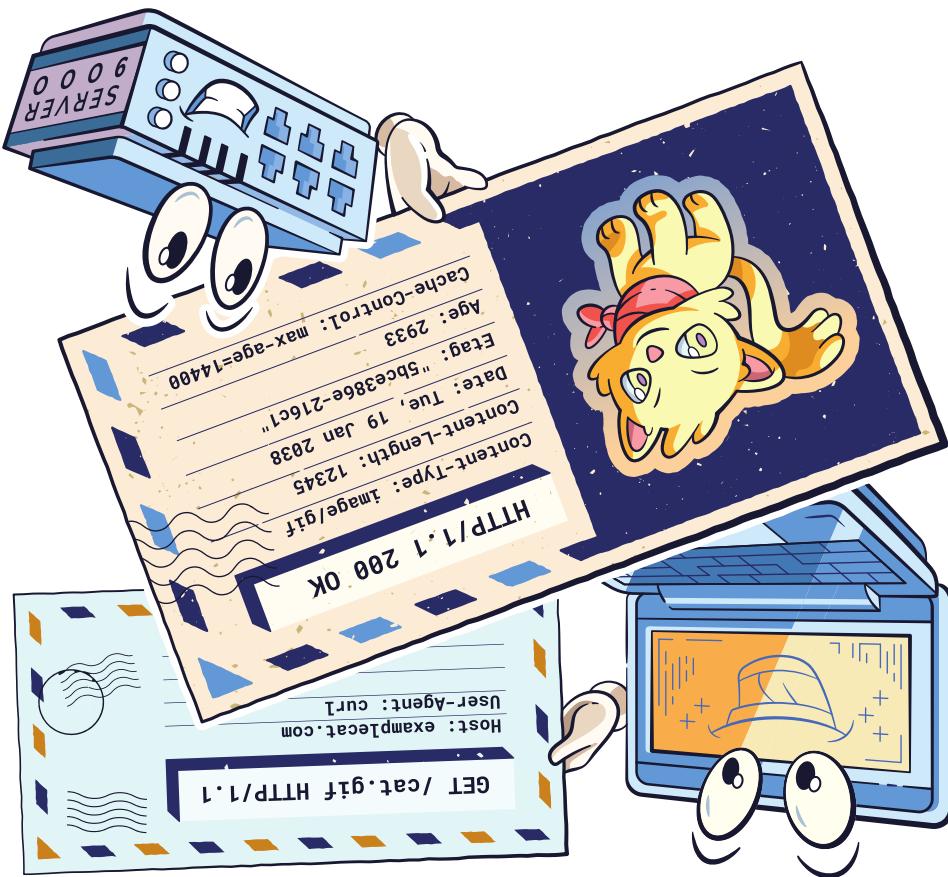


by Julia Evans



like this?
more zines at
wizardzines.com

about this zine

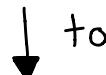
Your browser uses HTTP every time it visits a website.



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!

credits

Cover art: Vladimir Kašiković
Editing: Dolly Lanuza, Kamal Marhubi
Copy editing: Courtney Johnson
special thanks to Marco Rogers for
suggesting the idea of a HTTP zine

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> put RFC number here

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.



is the Host header
actually required?



Yes, section 5.4
of RFC 7230
says so!

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

Table of Contents

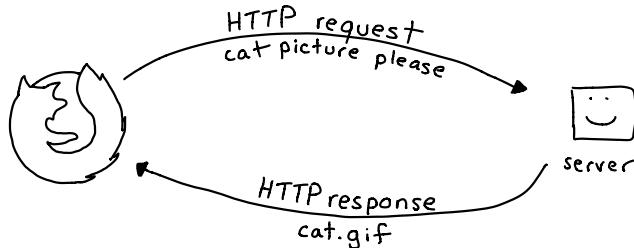
4	What's HTTP?
5	How URLs work
6	What's a header?
7	Anatomy of an HTTP request
8-9	Request methods (GET! POST!)
10	Request headers
11	Using HTTP APIs
12	Anatomy of an HTTP response
13	Responses
14	Status codes (200! 404!)
15	How cookies work
16-17	Content delivery networks & caching
18	Redirects
19	HTTP/2
20-21	HTTPS & certificates
22-24	Same origin policy & CORS
25	Security headers
26-27	Exercises & how to learn more

Making HTTP requests with curl to real internet websites and trying different headers is my favourite way to play around with HTTP & learn.	# curl tips
curl -i shows the response headers by sending a HEAD request	-i shows the response headers
curl -i https://examplecat.com/cat.txt -H "Range: bytes=8-17"	Try the Range header:
Request (and print out!) a compressed response:	curl -i https://examplecat.com
curl -i https://twitter.com -H "Accept-Language: es-ES"	Get a webpage in Spanish:
curl -i http://examplecat.com	curl -i http://examplecat.com
Get redirected to another URL:	Get redirected to another URL:
curl -i look at the Location header!	(hint: look at the Location header!)
curl -I https://github.com -H "Accept-Encoding: gzip" -- output -	curl -I https://github.com -H "Accept-Encoding: gzip" -- output -
Get a webpage in Spanish:	Get a webpage in Spanish:
curl -I http://twittter.com -H "Accept-Language: es-ES"	Guess what content delivery network GitHub is using:
curl -I https://gitub.githubbassets.com	(hint: it's in a header starting with X-)
curl -I example.com	Find out when example.com was last updated:
curl -I https://last-modified.com	(hint: Last-Modified)
curl -i examplecat.com/bananas	Get a 404 not found:
curl -i examplecat.com	Get a 404 not found:

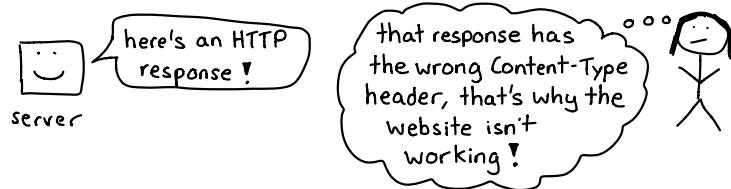
HTTP exercises

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 headers { Host: examplecat.com User-Agent: curl Accept: */*	{ GET / HTTP/1.1 status { HTTP/1.1 200 OK 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 Referer header. Example: Referrer-Policy: no-referrer.
spelling is inconsistent with Referer 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

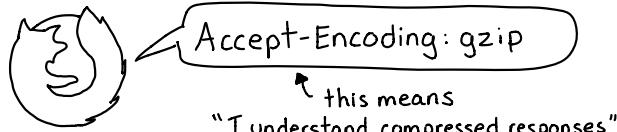
Certificate Transparency (CT) is a system that can help find malicious SSL certificates issued for your site. This header gives the browser a URL to use to report bad certificates to you.

X-XSS-Protection

Another way to protect against XSS attacks. It's not supported by all browsers, and Content-Security-Policy is more powerful.

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.

Accept-Encoding: gzip
↑ name ↑ 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
Range: bytes=1-10

Accept-Encoding: gzip
Accept-Language: es-ES

Every Accept-header has a corresponding Content- header

Manage caches:

ETag: "abc123"
If-None-Match: "abc123"
Vary: Accept-Encoding
If-Modified-Since: 3 Aug 2019 13:00:00 GMT
Last-Modified: 3 Feb 2018 11:00:00 GMT
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)

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. So what's the big deal?

Here are 2 reasons it's important to prevent Javascript code from making arbitrary requests from your browser:

Reason 1: cookies

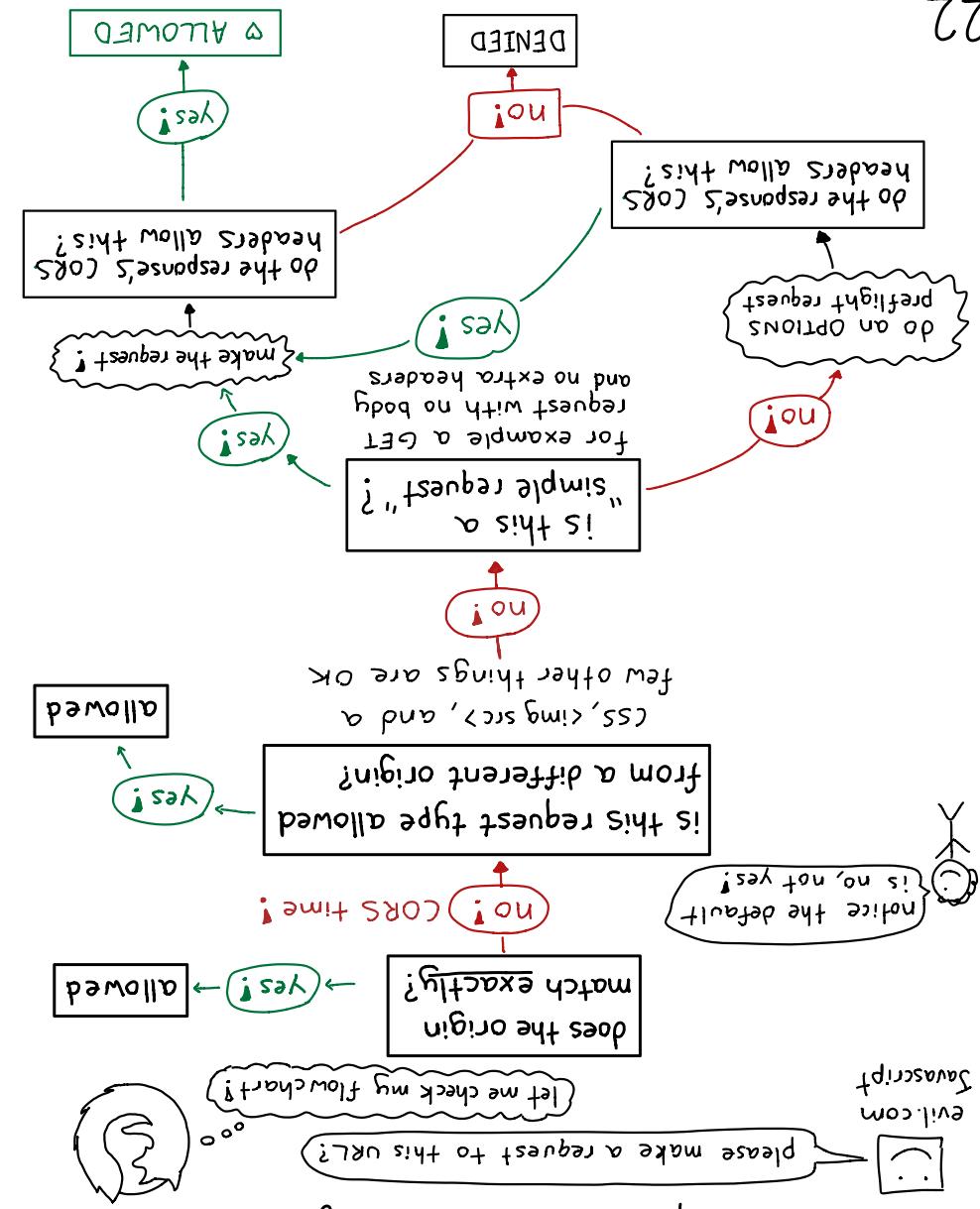
Browsers often send your cookies with HTTP requests. You don't want evil.com to be able to make requests using your login cookies. They'd be logged in as you!



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 is one way browsers protect you from malicious JavaScript code. Here's basically how it works:

example: `https://fabby.example.cat.com:443`

An origin is the protocol + domain including subdomains + port.

malicious JavaScript code makes a request to this URL:

`evil.com`

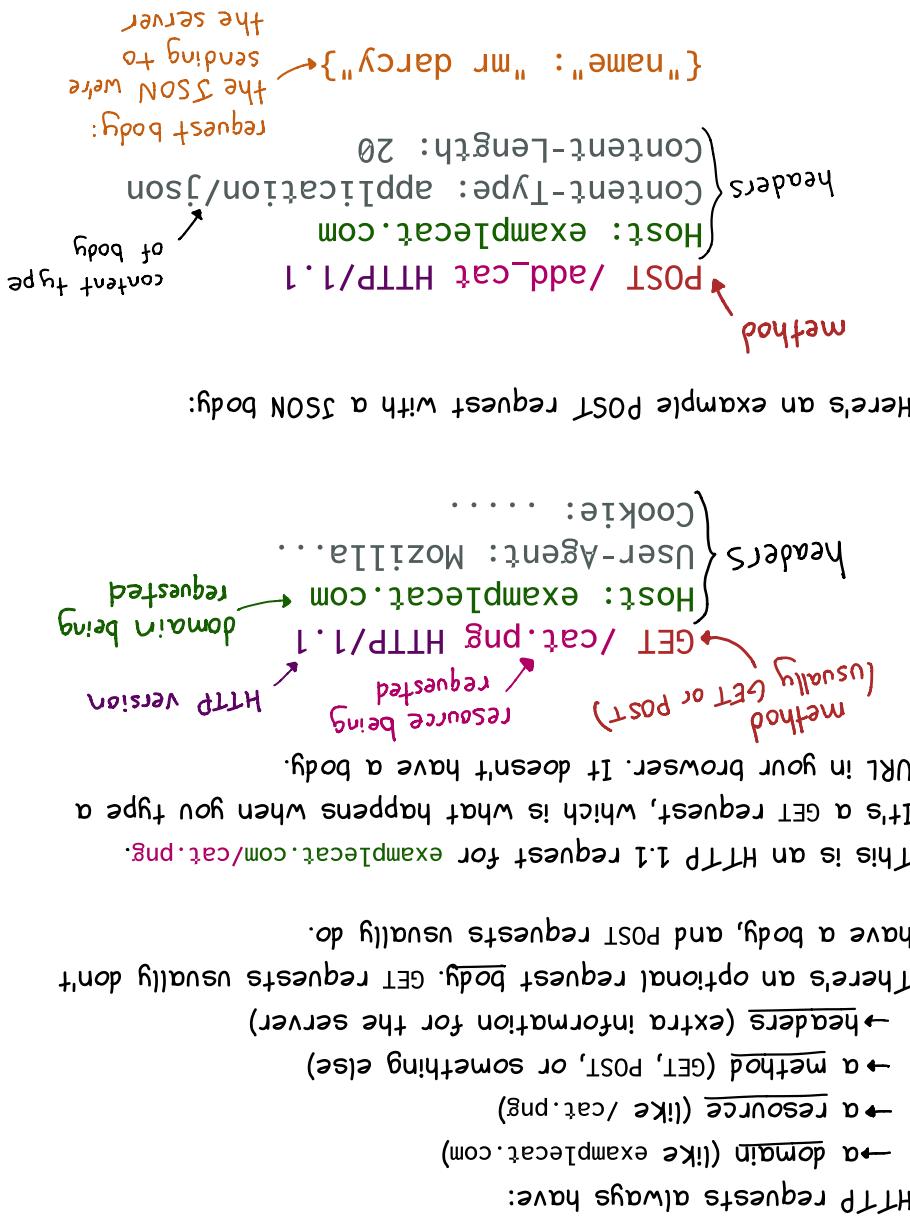
`please make a request to this URL?`

`let me check my flavor chart?`

`evil.com`

`JavaScript`

The same origin policy



* HTTP requests *

anatomy of an

request methods

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

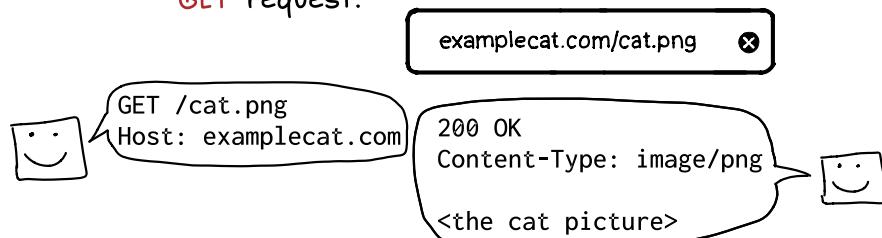
this means it's a GET request

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).

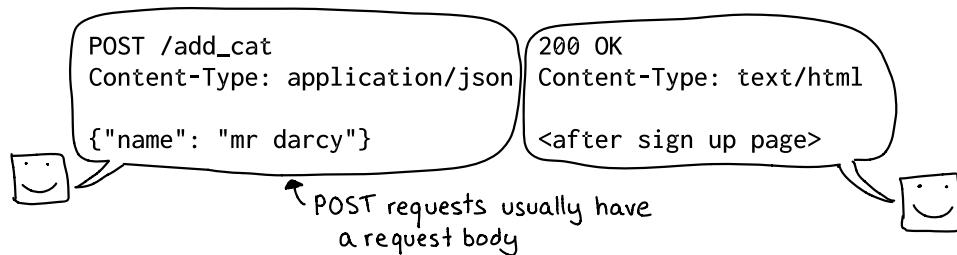
GET

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



POST

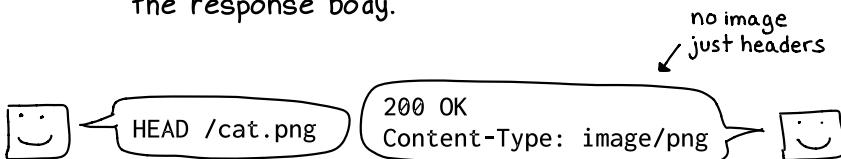
When you hit submit on a form, that's (usually) a POST request.



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

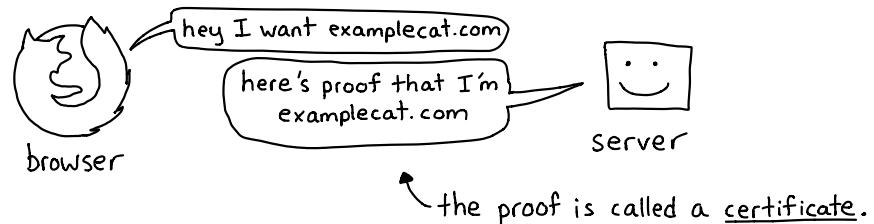
HEAD

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



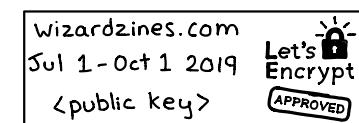
certificates

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 that only the server has *this is the only secret part, the rest is public*
- a public key to use when encrypting
- a cryptographic signature from someone trusted

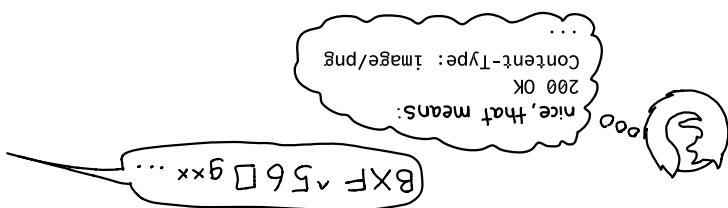


The trusted entity that signs the certificate is called a ★ Certificate Authority ★ (CA) and they're responsible for verifying that you actually own the domain:

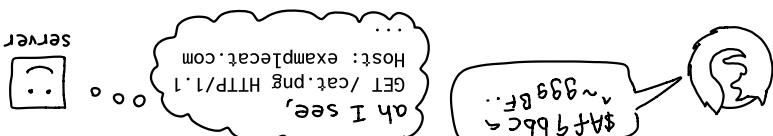


When your browser connects to examplecat.com, it validates the certificates using a list of trusted CAs installed on your computer. These CAs are called "root certificate authorities".

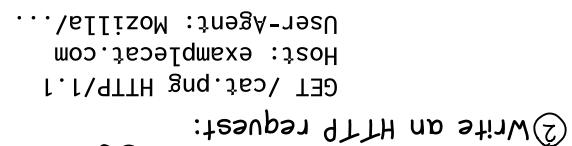




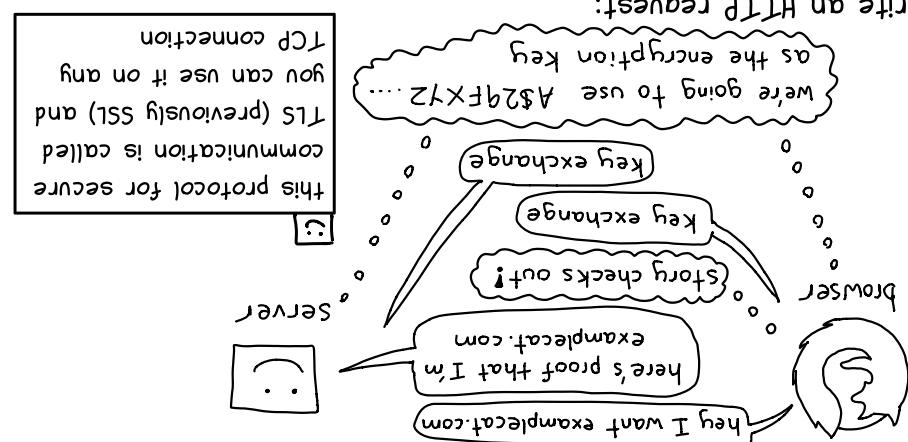
④ Receive encrypted HTTP responses:



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



② Write an HTTP request:



simplified version of how picking the encryption key works:

server use the same key to encrypt/decrypt content. Here's a
use for this connection to examplecat.com. The browser and

① Negotiate an encryption key (AES symmetric key) to

<https://examplecat.com/cat.png>:

Here's what your browser does when it asks for

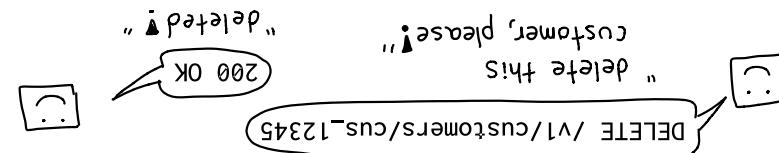
HTTPs: HTTP + Secure ⚡

OPTIONS

OPTIONS is mostly used for CORS requests.
Page 24 has more about CORS.
It also tells you which methods are available.

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

DELETE



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

PUT

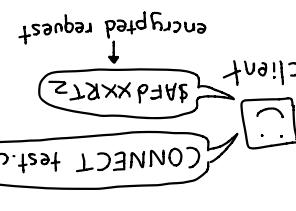
Used in some APIs (like the Stripe API)
to partial updates to a
resource ("just change this file").

PATCH

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

CONNECT

Different from the others: instead of making
a request directly to a server, 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: curl 7.0.2 Referer: https://examplecat.com

User-Agent

name + version of your browser and OS

User-Agent: curl 7.0.2 Referer: https://examplecat.com

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: Wed, 21 Oct...

If-Modified-Since

only send if resource was modified after this time

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)

method: GET

path: /cat.gif

headers:

- Host: examplecat.com
- User-Agent: curl

after (HTTP/2)

method: GET

path: /cat.gif

authority: examplecat.com

- headers:
- 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)

→ request 1

response 1 ←

→ request 2

response 2 ←

after (HTTP/2)

→ request 1

→ request 2

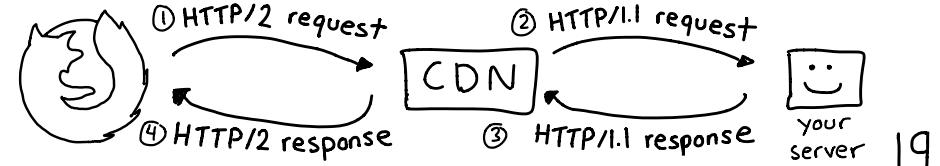
out of order is ok { response 2 ←
response 1 ← }

} one TCP connection

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

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

Set the right Content-Type header

- * application/json → JSON;
- * application/x-www-form-urlencoded → same as what a HTML form does
- * application/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)

Most HTTP APIs require a secret API key so they know who you are.

https://api.twilio.com/2010-04-01/Accounts/ACCOUNT_ID/Messages.json

-d '{ "FROM": "+15141234567",
-u sends the user name/pattern word

this sends a POST request

Warning!

your site to a new domain!

Setting up redirections is a great thing to do if you move examples/earcater.com/purple_cat_teaches_ceti_to_purple_droids.com

The `Location` header tells the browser what new URL to use. The new URL doesn't have to be on the same domain:

The diagram illustrates a sequence of two HTTP interactions between a browser and a server.

Browser sends a `GET /cat.png HTTP/1.1` request to the `Host: example.cat.com`.

Server responds with a `301 Moved Permanently` status, indicating the resource has moved to `/cat.png`. The **Browser** notes this and sends a new `GET /cat.png HTTP/1.1` request to the `Host: example.cat.com`.

Server responds with a `200 OK` status, and the **Browser** displays the content with the message `<rest of website here>`.

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

but you end up at a slightly different URL, like this:


 examplecat.com/dog.png

Sometimes you type a URL into your browser like this:

redirects

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 status
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

\   / cat!
 ) ( ' ) 
( / ) 
\(_)_|
```

status code

headers

body

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

- and more:

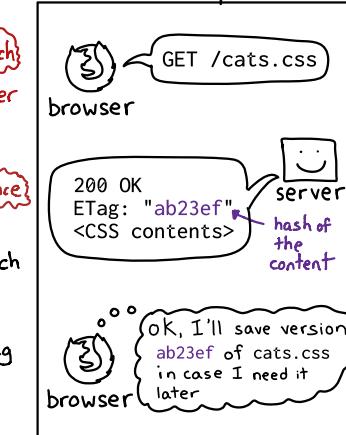
```

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

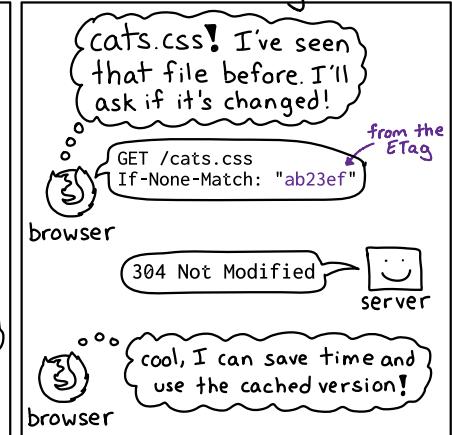
caching headers

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

initial request



the next day



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	fjxx99cef.. (compressed gibberish)

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

Cache-Control

request AND response header

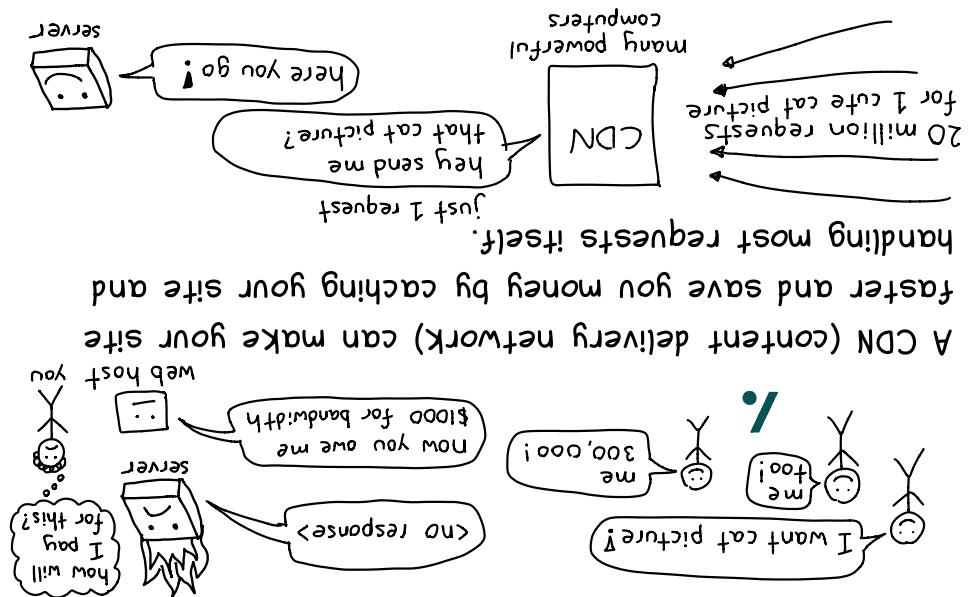
Used by both clients and servers to control caching behaviour. For example:

Cache-Control: max-age=999999999999 from the server asks the CDN or browser to cache the thing for a long time.

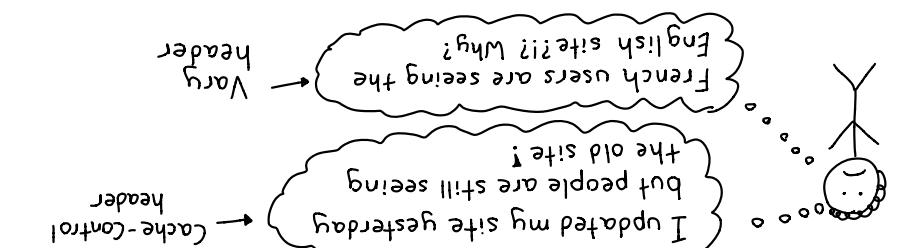
In 2004, if your website suddenly got popular, often the webserver wouldn't be able to handle all the requests.

Content delivery networks

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

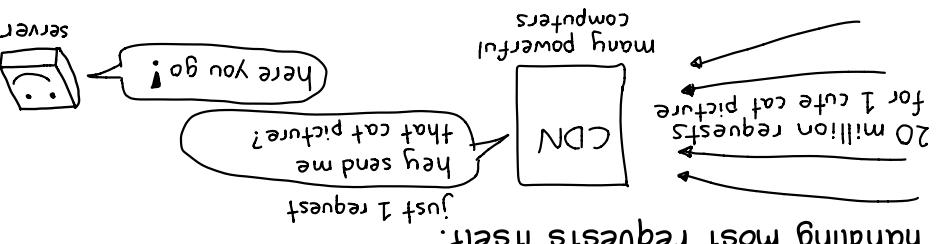


Next, we'll explain the HTTP headers your CDN or browser uses to decide how to do caching.



This is great but caching can cause problems too!

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



Browser uses to do caching.

Whether Range requests header is supported for this resource

Accept - Ranges

Whether body is compressed
Content-Encoding: gzip

Content-Encoding

Length of body in bytes
Content-Length: 33

Content-Length

Allows cross-origin requests.
Cross-Origin Resource Sharing (CORS) headers. These

Access-Control *

Keep-alive
TCP connection open after this time.

Connection

HTTP based on what response will
request headers

Vary

When content was last modified
(not always accurate)

Cache-Control

When response was sent Date: Mon, 09 Sep 2019...
How many seconds response has been cached Age: 355

Last-Modified

Age

response headers

URL to redirect to
Location: /cat.png

Location

Language of body
Content-Language: en-US

Content-Language

MIME type of body
Content-Type: text/plain

Content-Type

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

Set-Cookie

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

Expires

Cache-Control: max-age=300
Response body

Via

Proxy servers
added by
via: nginx

ETag

Etag: "ac5afffa."
Version of response body

Age

When response was sent Date: Mon, 09 Sep 2019...
How many seconds response has been cached Age: 355

Webserver wouldn't be able to handle all the requests.

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

HTTP status codes

Every HTTP response has a ★status code★



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

200 OK

} 2xx's mean
★ success ★

301 Moved Permanently

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

302 Found
temporary redirect

304 Not Modified
the client already has the latest
version, "redirect" to that

} 4xx errors
generally mean
the client made
an invalid request

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

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

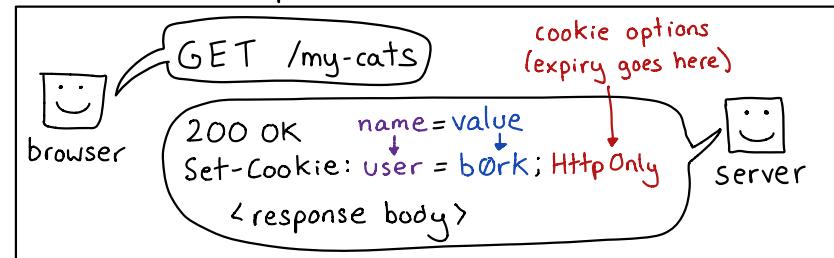
504 Gateway Timeout
the server was too slow to respond

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".