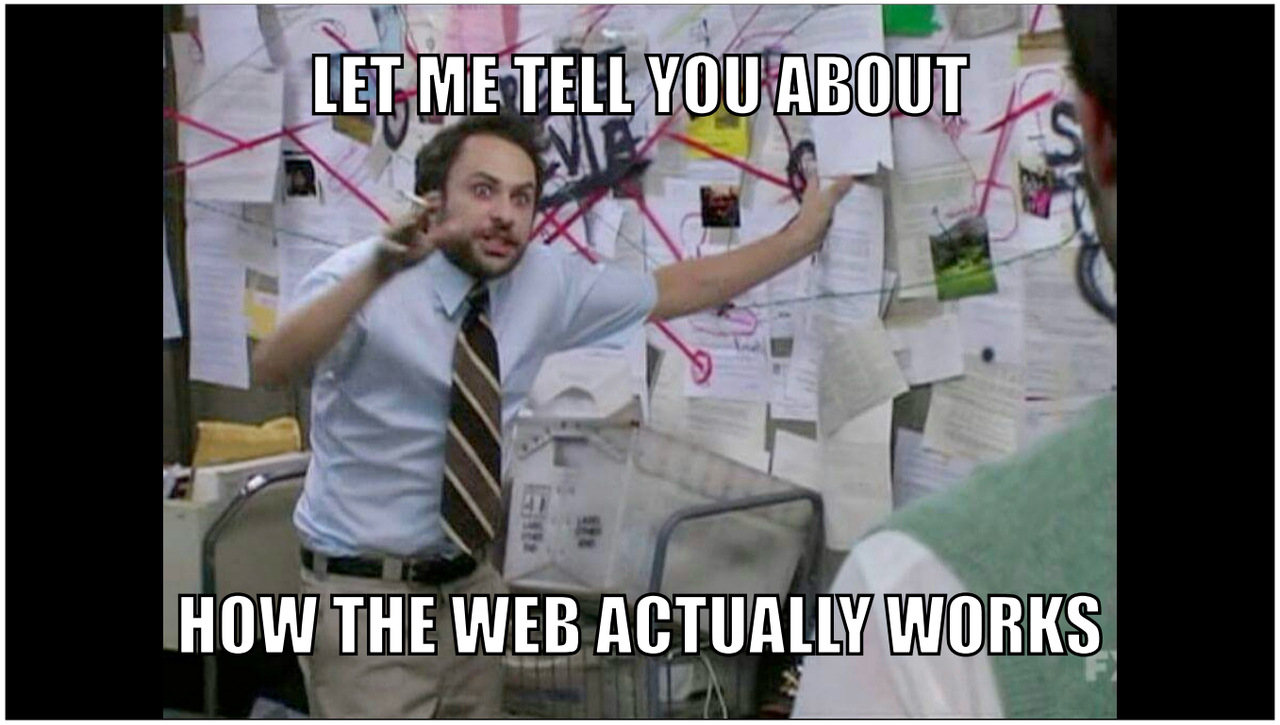
Web Architecture 101

How web sites work

1. HTTP
2. Front end vs back end
3. The front end
4. The back end
5. Data and APIs
6. More complex setups


LET ME TELL YOU ABOUT
HOW THE WEB ACTUALLY WORKS

What happens when you enter a URL into a web browser?

[this section covers the anatomy of a URL, DNS, and HTTP]

I have used this as an interview question before.

Much like a child who responds to every answer with "but why?", there isn't a single right answer. How deep do you want to go? Well, let's make a start.

Front End Back End

Client URL Server

Content-Type

HTTP DNS HTTPS

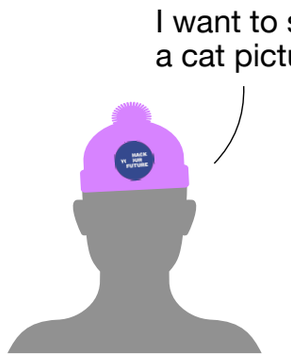
Perhaps a show of hands for who feels that they know what these are.

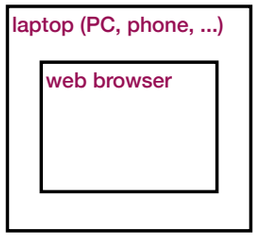
How to see pictures of cats

You have your computer (laptop, phone, whatever), with a working Internet connection, and you want to see a picture of a cat. You already know the exact URL. So you go to your web browser and type in the URL.

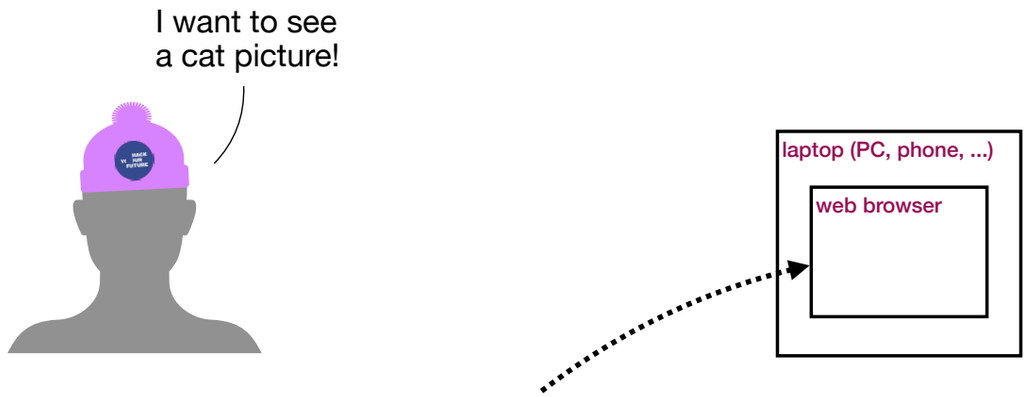
Your web browser is just software. So what does it do with what you typed in?

How to see pictures of cats

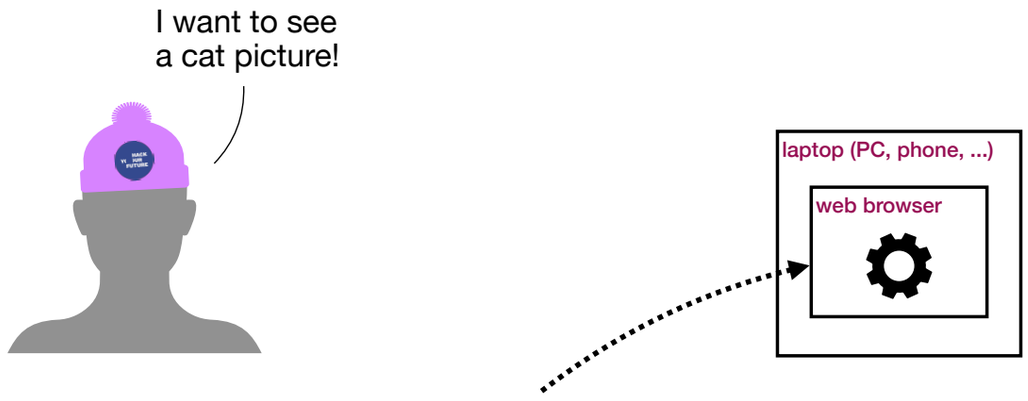




How to see pictures of cats



How to see pictures of cats



Understanding URLs

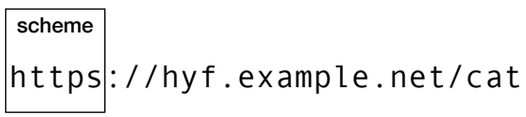
```
https://hyf.example.net/cat
```

The authority is usually just a server name, but *can* include a port number, and *can* include a username and password.

The path *can* be followed by a query ("?..."). The whole thing *can* be followed by a fragment ("#...").

Understanding URLs

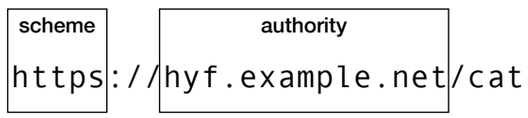
- The part up to the ":" is called the *scheme*. It tells the browser how to interpret the rest of the URL, and also how to actually go about fetching the resource at that URL.



Understanding URLs

- The part up to the ":" is called the *scheme*. It tells the browser how to interpret the rest of the URL, and also how to actually go about fetching the resource at that URL.
- The part from "//" to the next "/" is called the *authority*.

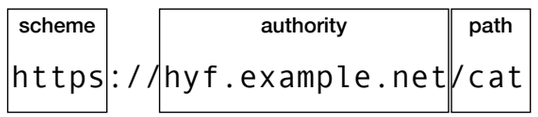
less formally, this is usually known as the "host", "server", "site", or "domain"

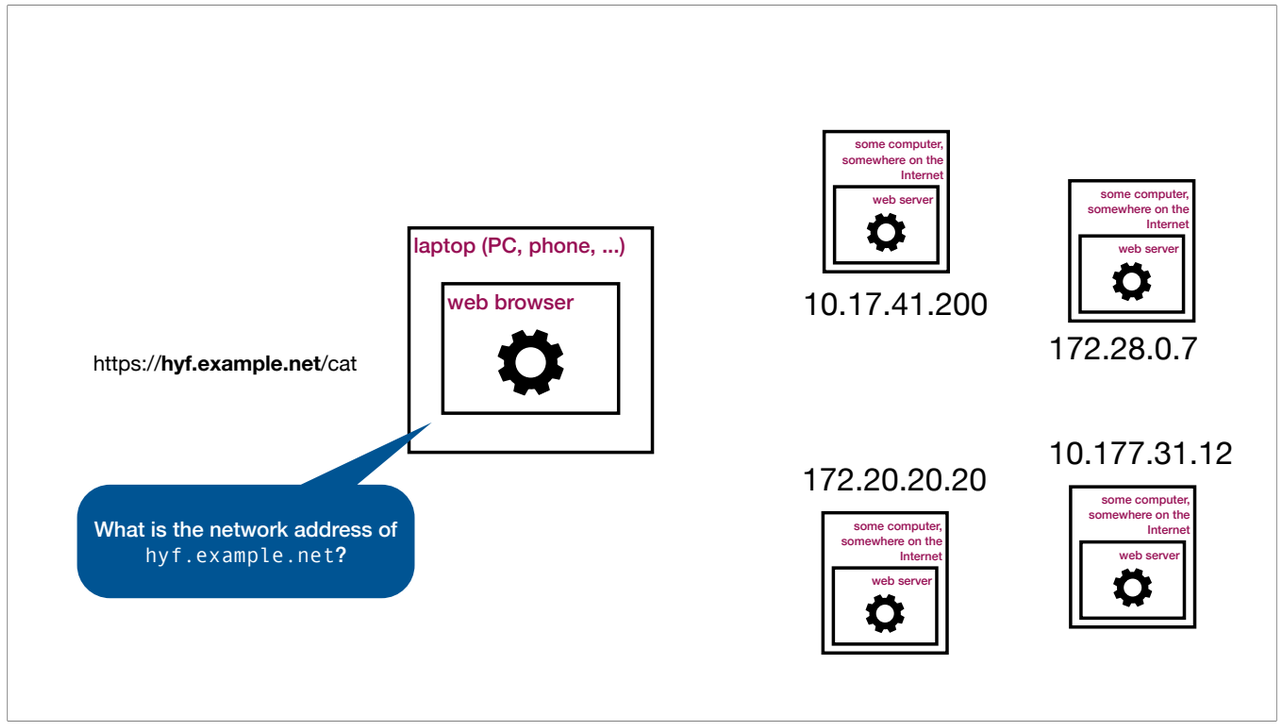


Understanding URLs

- The part up to the ":" is called the *scheme*. It tells the browser how to interpret the rest of the URL, and also how to actually go about fetching the resource at that URL.
- The part from "//" to the next "/" is called the *authority*.
- The part from that "/" onwards is called the *path*.

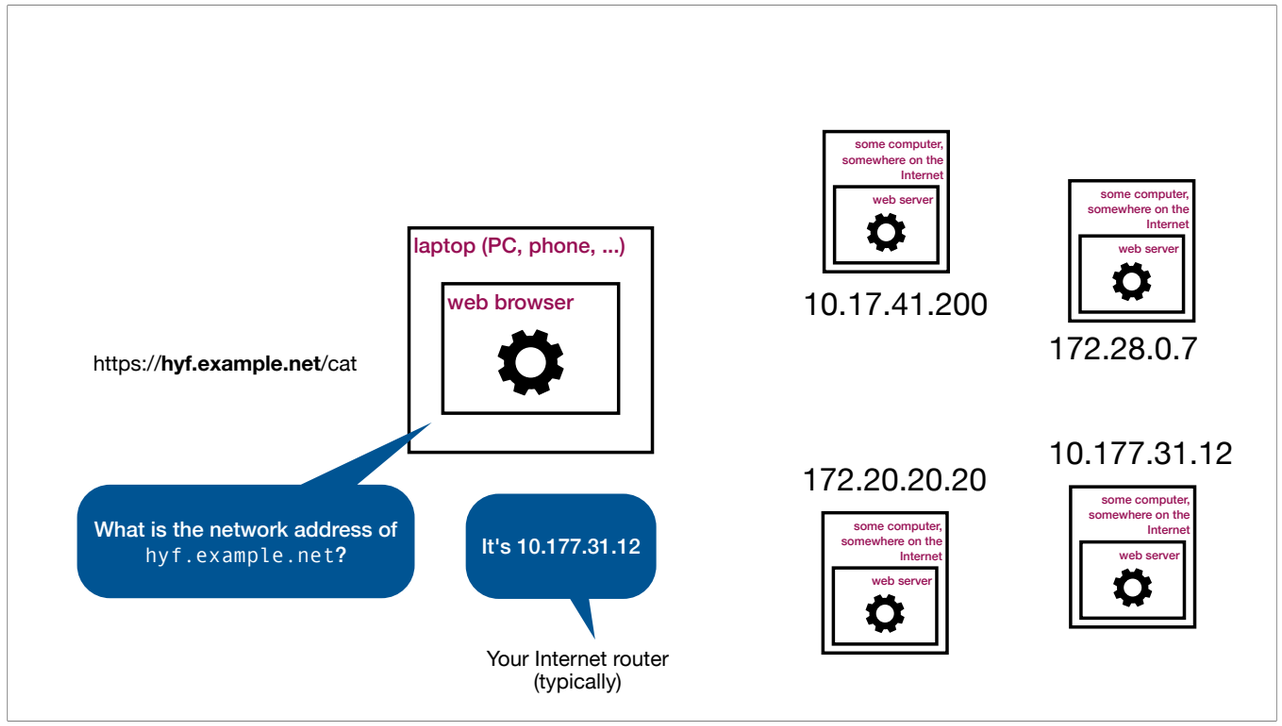
less formally, this is usually known as the "host", "server", "site", or "domain"





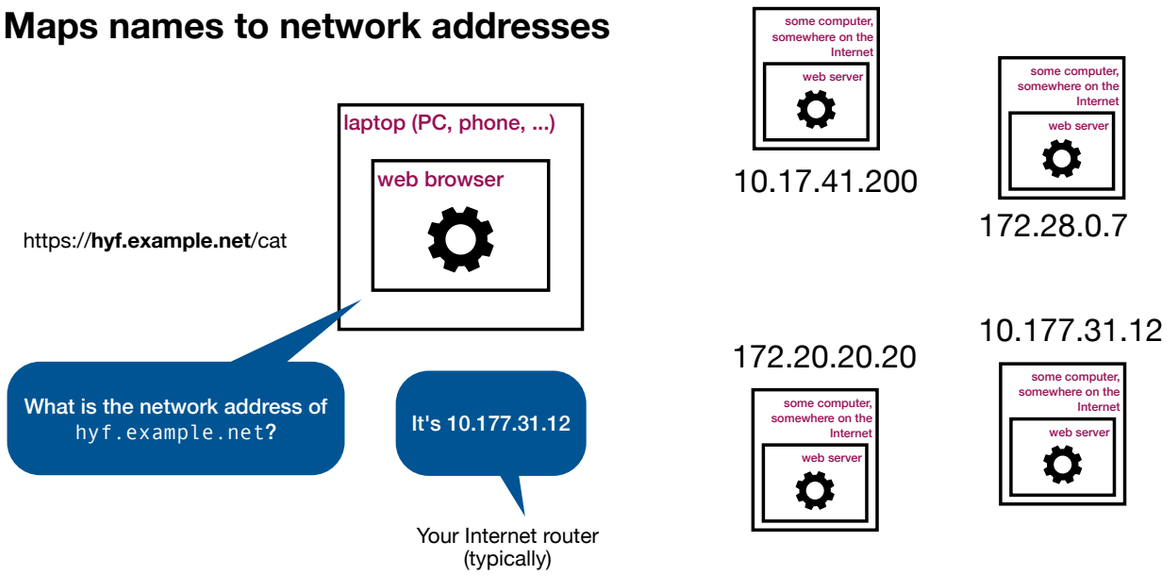
To find out, your computer asks (typically) your Internet router "Which computer is hyf.example.net?" — since that's the "host" part of the URL in question. The router replies with the answer.

This is what we call DNS: the Domain Name System.


some computer, somewhere on the Internet
web server
10.17.41.200
some computer, somewhere on the Internet
web server
172.28.0.7
laptop (PC, phone, ...)
web browser
https://hyf.example.net/cat
What is the network address of hyf.example.net?
It's 10.177.31.12
Your Internet router (typically)
172.20.20.20
some computer, somewhere on the Internet
web server
10.177.31.12
some computer, somewhere on the Internet
web server

DNS

Maps names to network addresses

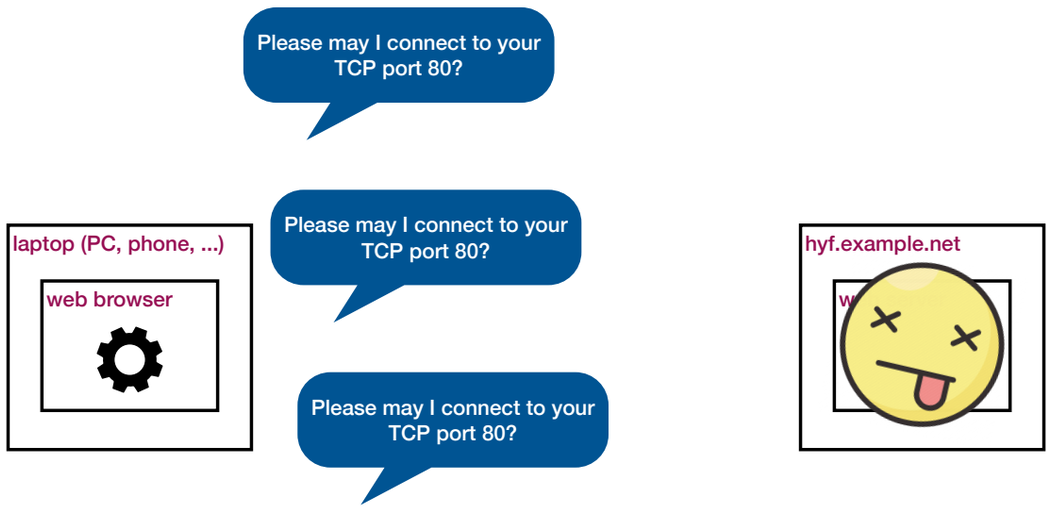


Connecting

Negotiating a connection. Always exactly two ends: "client" and "server". Server listens and waits; client initiates a connection to the server.

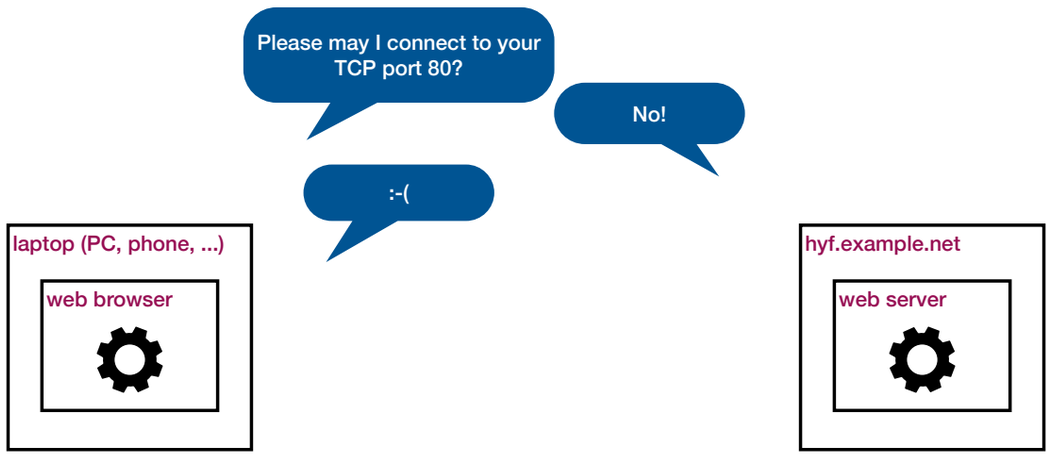
Why port 80? It's the standard port number if the scheme is "http"."https", on the other hand, uses port 443. It uses a encrypted connection, not an unencrypted one. Otherwise, it's the same.

Connection timed out



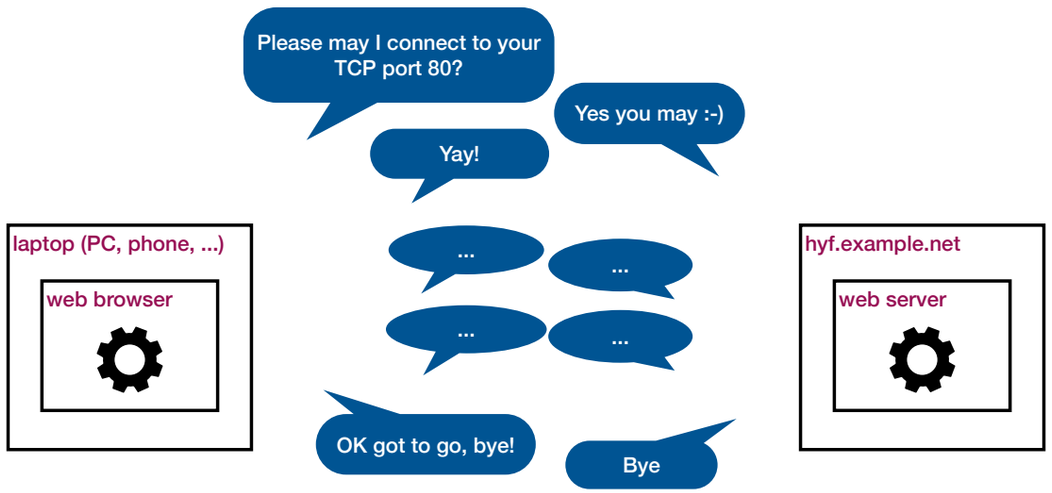
Connection timed out / EUNREACH

Connection refused



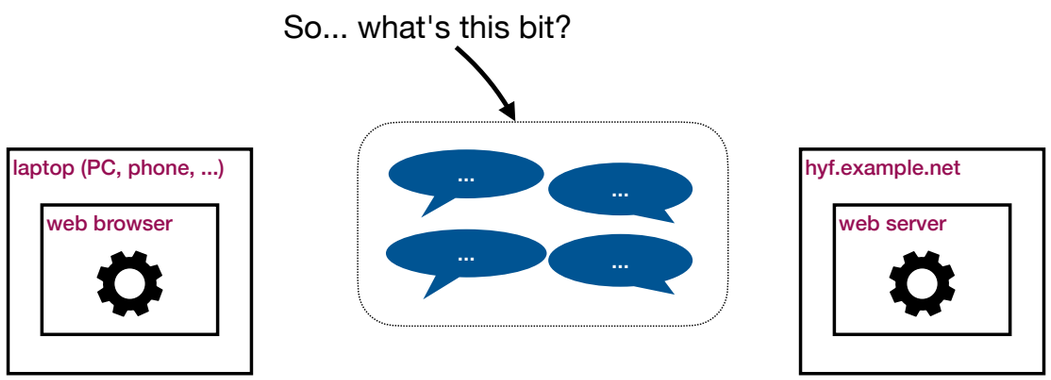
Connection refused / ECONNREFUSED

Connecting

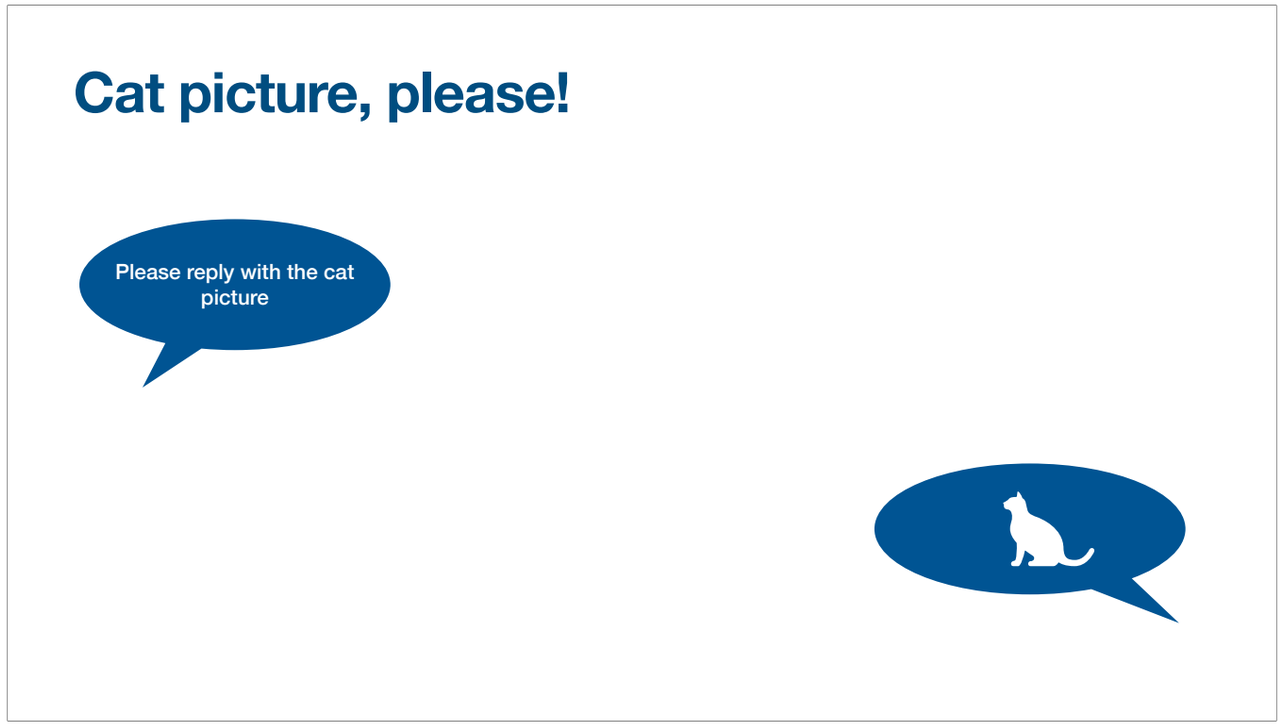


(this slide only here to provide a nice transition to the next one)

Connecting



The connection allows the client and server to talk to each other. What do they say?



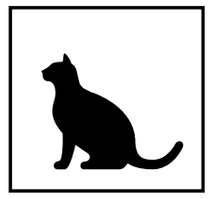
Once a connection has been established, the client and server can talk to each other. So what do they say?

What is a cat?

Filename: cat.jpg

Content:

```
00000000: ffd8 ffe0 0010 4a46 4946 0001 0100 0001  ......JFIF......
00000010: 0001 0000 ffdb 0043 0006 0405 0605 0406  .......C........
00000020: 0605 0607 0706 080a 100a 0a09 090a 140e  ................
00000030: 0f0c 1017 1418 1817 1416 161a 1d25 1f1a  .............%..
00000040: 1b23 1c16 1620 2c20 2326 2729 2a29 191f  .#... , #&')*)..
00000050: 2d30 2d28 3025 2829 28ff db00 4301 0707  -0-(0%()(...C...
00000060: 070a 080a 130a 0a13 281a 161a 2828 2828  ........(...((((
00000070: 2828 2828 2828 2828 2828 2828 2828 2828  ((((((((((((((((
00000080: 2828 2828 2828 2828 2828 2828 2828 2828  ((((((((((((((((
00000090: 2828 2828 2828 2828 2828 2828 2828 ffc0  ((((((((((((((..
000000a0: 0011 0800 4800 8003 0122 0002 1101 0311  ....H....".....
000000b0: 01ff c400 1f00 0001 0501 0101 0101 0100  ................
000000c0: 0000 0000 0000 0001 0203 0405 0607 0809  ................
000000d0: 0a0b ffc4 00b5 1000 0201 0303 0204 0305  ................
000000e0: 0504 0400 0001 7d01 0203 0004 1105 1221  ......}........!
000000f0: 3141 0613 5161 0722 7114 3281 91a1 0823  1A..Qa."q.2....#
00000100: 42b1 c115 52d1 f024 3362 7282 090a 1617  B...R..$3br.....
                          ... (etc)
```



Well, what is a cat [picture]?

On your computer, a photographic picture is often saved as a JPEG file — something dot J P G.
Inside that file, there's ... stuff. Data. Bytes.

What is a cat?

Content-Type: image/jpeg

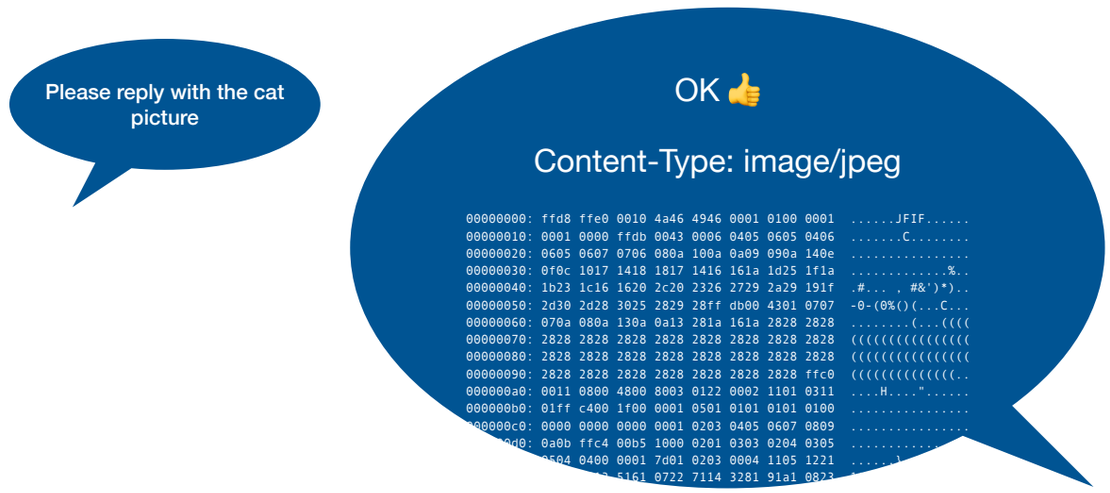
Content:

```
00000000: ffd8 ffe0 0010 4a46 4946 0001 0100 0001  ......JFIF......
00000010: 0001 0000 ffdb 0043 0006 0405 0605 0406  .......C........
00000020: 0605 0607 0706 080a 100a 0a09 090a 140e  ................
00000030: 0f0c 1017 1418 1817 1416 161a 1d25 1f1a  .............%..
00000040: 1b23 1c16 1620 2c20 2326 2729 2a29 191f  .#... , #&')*)..
00000050: 2d30 2d28 3025 2829 28ff db00 4301 0707  -0-(0%()(...C...
00000060: 070a 080a 130a 0a13 281a 161a 2828 2828  ........(...((((
00000070: 2828 2828 2828 2828 2828 2828 2828 2828  ((((((((((((((((
00000080: 2828 2828 2828 2828 2828 2828 2828 2828  ((((((((((((((((
00000090: 2828 2828 2828 2828 2828 2828 2828 ffc0  ((((((((((((((..
000000a0: 0011 0800 4800 8003 0122 0002 1101 0311  ....H....".....
000000b0: 01ff c400 1f00 0001 0501 0101 0101 0100  ................
000000c0: 0000 0000 0000 0001 0203 0405 0607 0809  ................
000000d0: 0a0b ffc4 00b5 1000 0201 0303 0204 0305  ................
000000e0: 0504 0400 0001 7d01 0203 0004 1105 1221  ......}........!
000000f0: 3141 0613 5161 0722 7114 3281 91a1 0823  1A..Qa."q.2....#
00000100: 42b1 c115 52d1 f024 3362 7282 090a 1617  B...R..$3br.....
                          ... (etc)
```

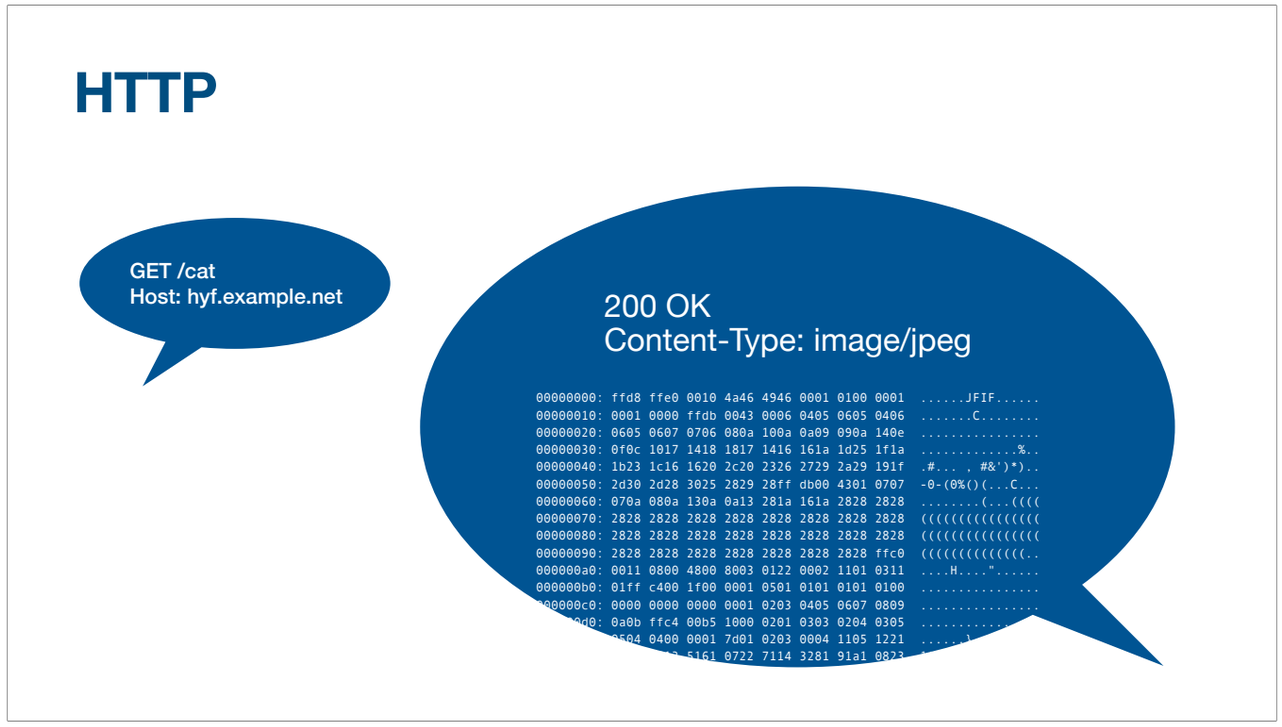
On the Internet, *filenames* aren't so important. Instead, we talk about "content types". A content type answers the question, "How should I interpret all of these bytes?".

The content type for JPEG images is "image/jpeg".

Cat picture, please!



So if you ask for a picture of a cat pic, then the answer looks like this. OK, content-type, data.

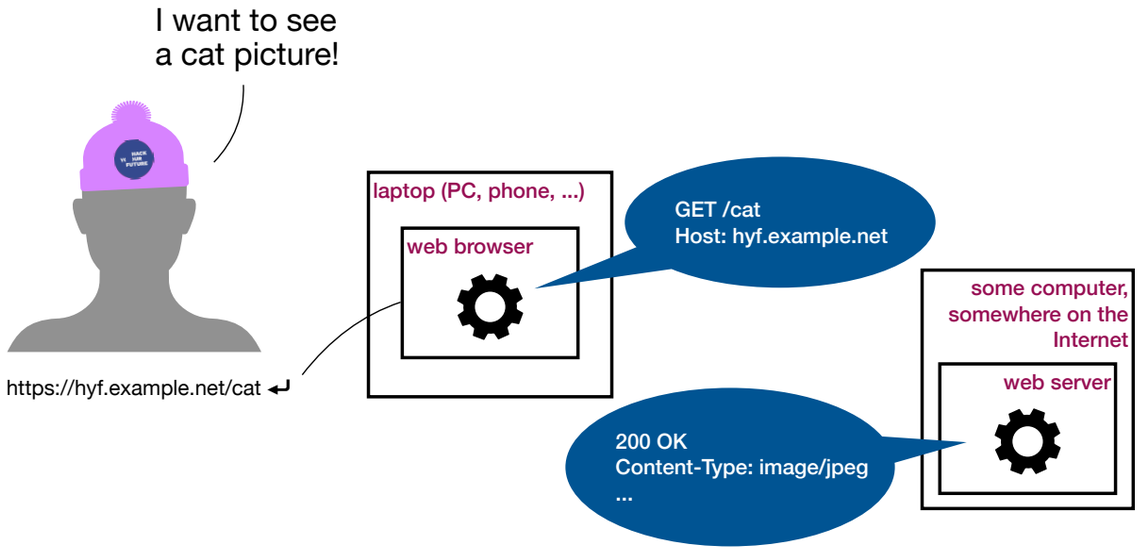


To turn that into the language that computers actually use to communicate:

"Please reply with the cat picture" becomes the pictured "GET" request. "/cat" is the path from the URL.
"OK here you go" becomes a "200 OK" response.

This is HTTP. The somewhat-anachronously-named "Hypertext Transfer Protocol".

How to see pictures of cats on the Internet



Cat tax.

How to see the content-type / page source

Chrome: ?

Firefox: cmd-I "Page Info"

Edge: ?

Safari: ?

A simple game

A simple game

Let's say you've created a game, to be played in a web browser.

It doesn't really matter here what the game *does*. Maybe it's "guess what number I'm thinking of" or something. Anything.

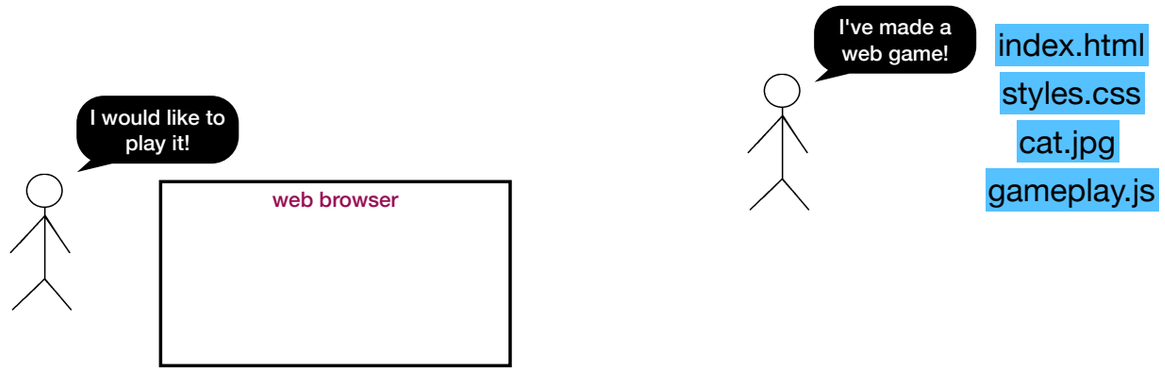
The point is, you've made a page.

A simple game

Here are the files which make up your game-page:

index.html
styles.css
cat.jpg
gameplay.js

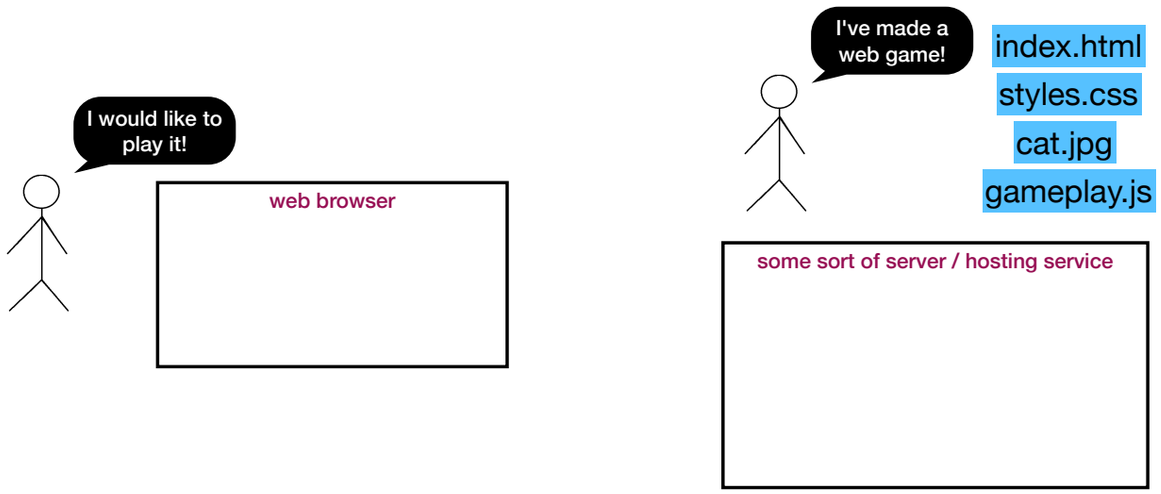
How can I play your game?



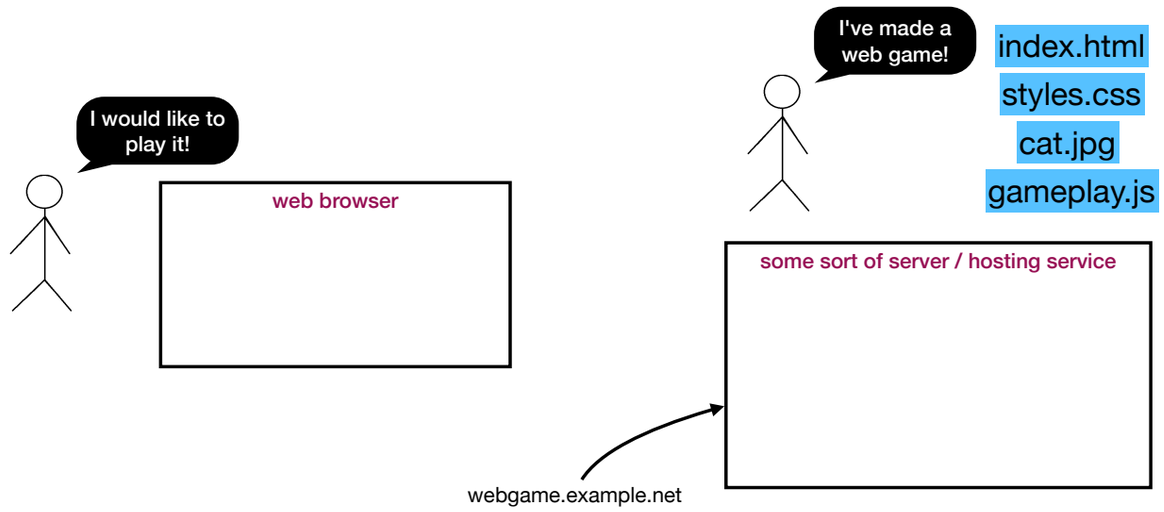
What do we need?

- a server
- server has to have a name

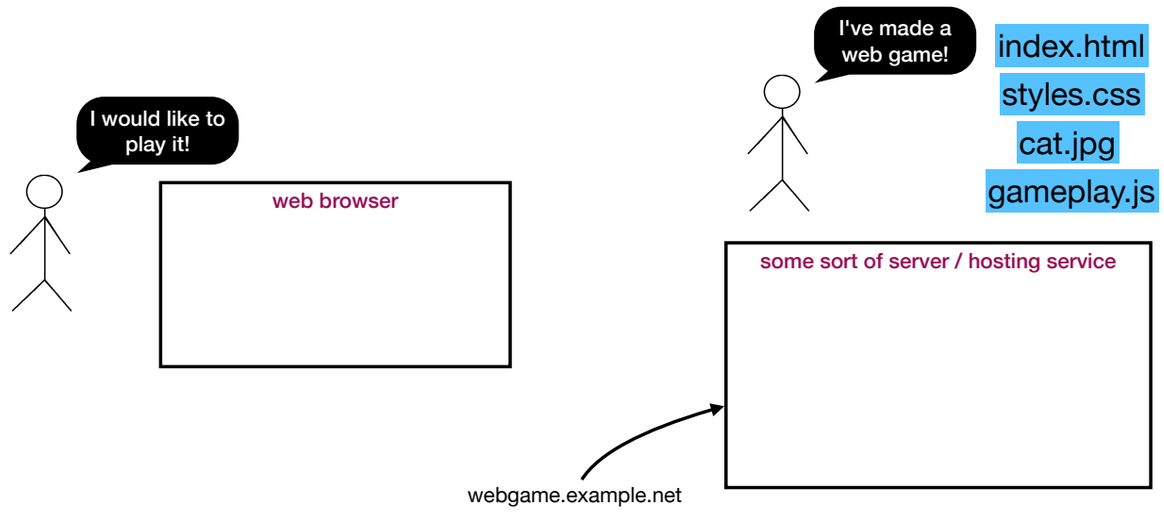
How can I play your game?



How can I play your game?



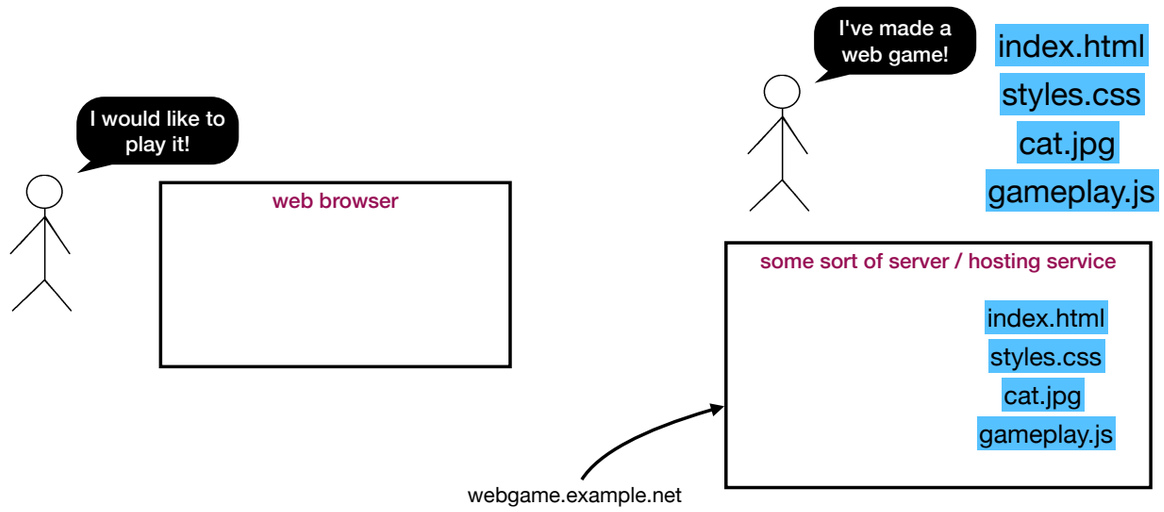
How can I play your game?



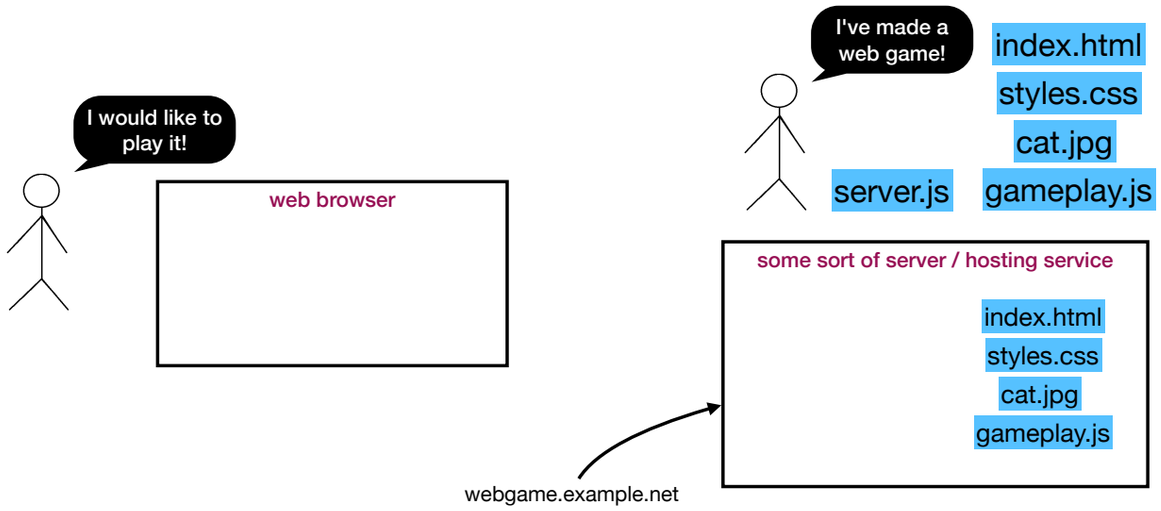
What next?

- the game files need to be on the server
- we need a webserver, to actually make them available. write it, deploy it, run it.

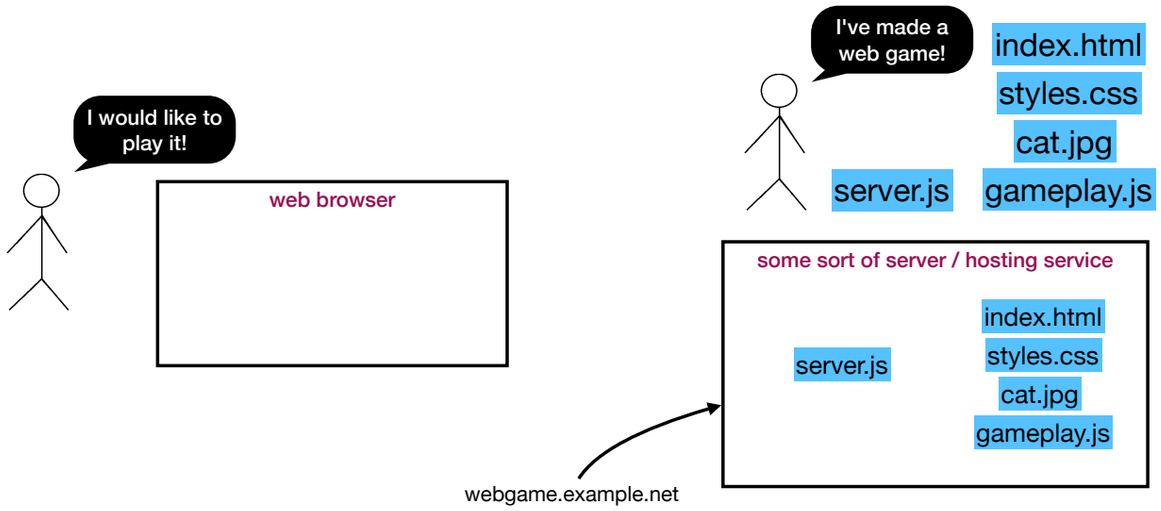
How can I play your game?



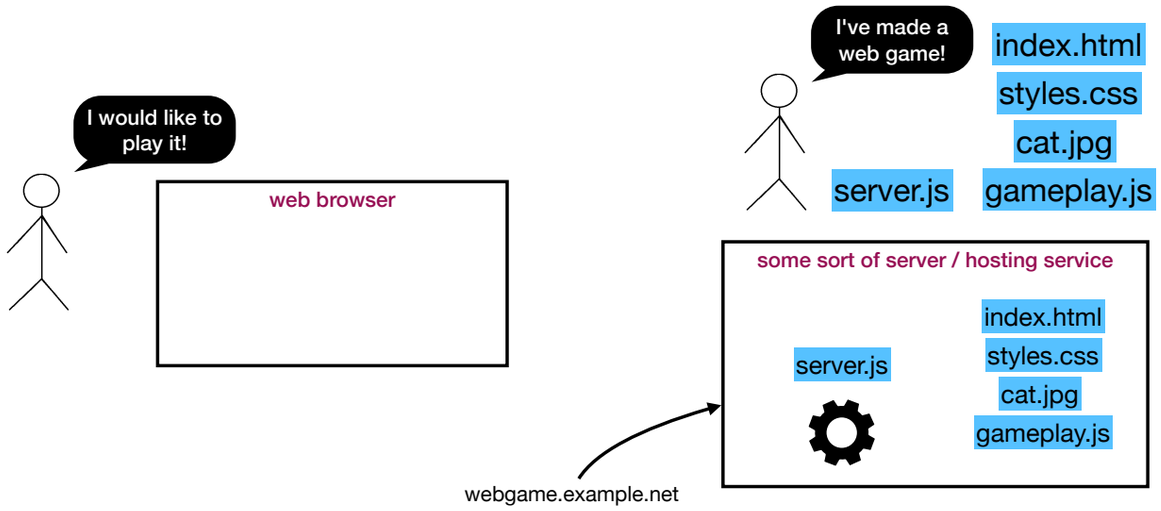
How can I play your game?



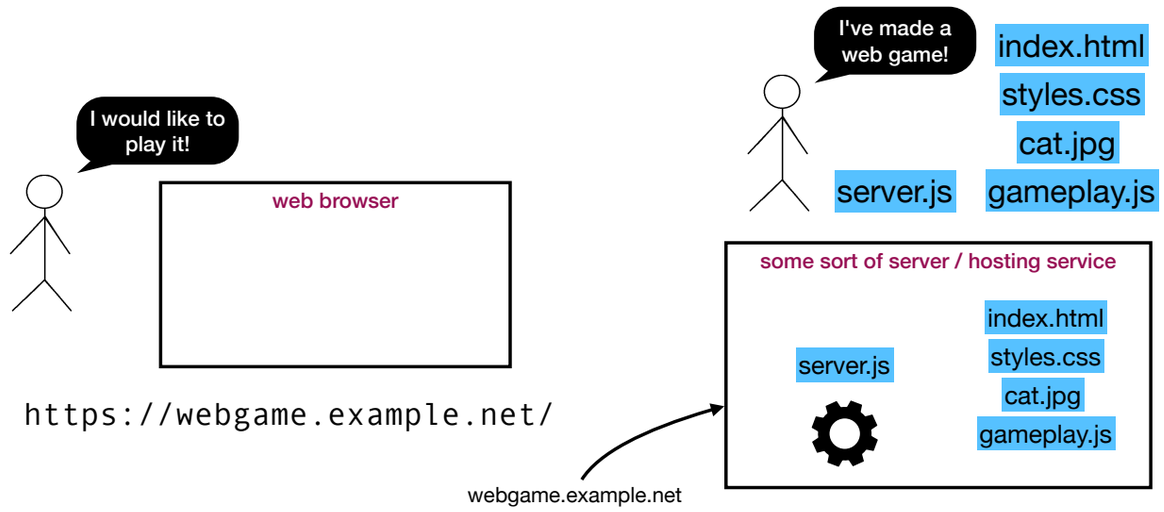
How can I play your game?



How can I play your game?

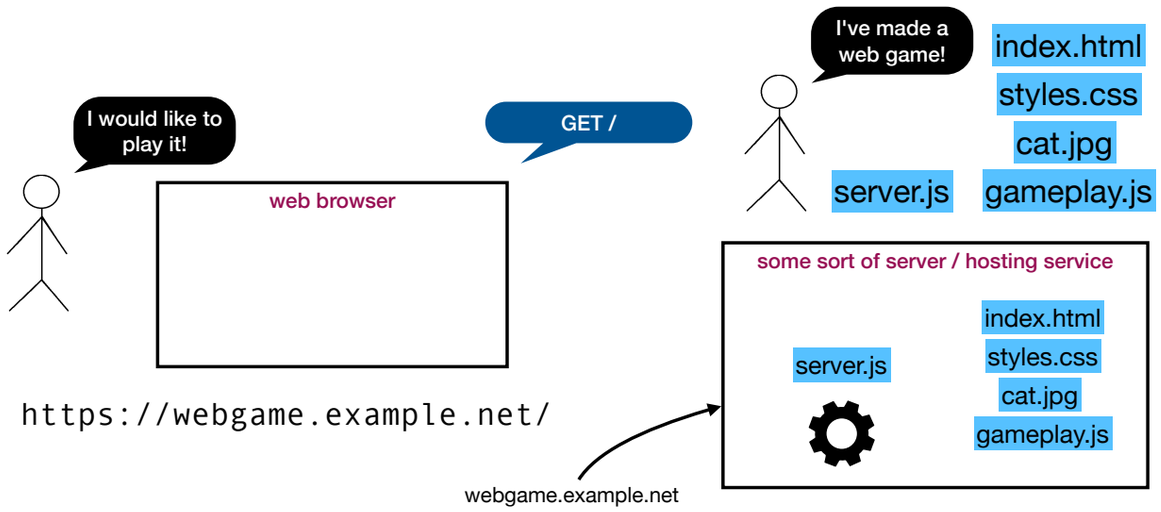


How can I play your game?

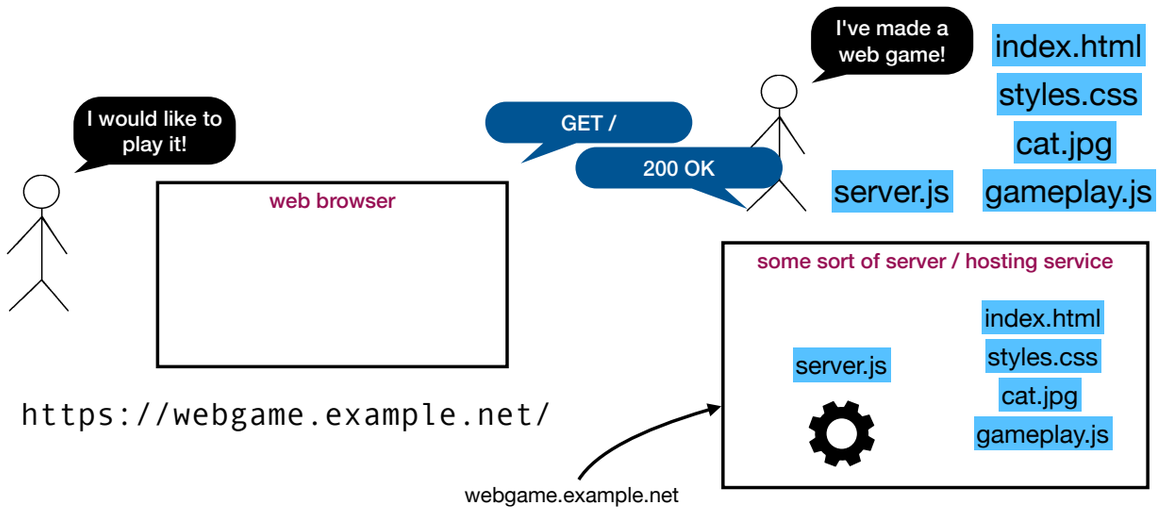


It's ready! Here's the URL.
Player's web browser fetches the html, then the other stuff.

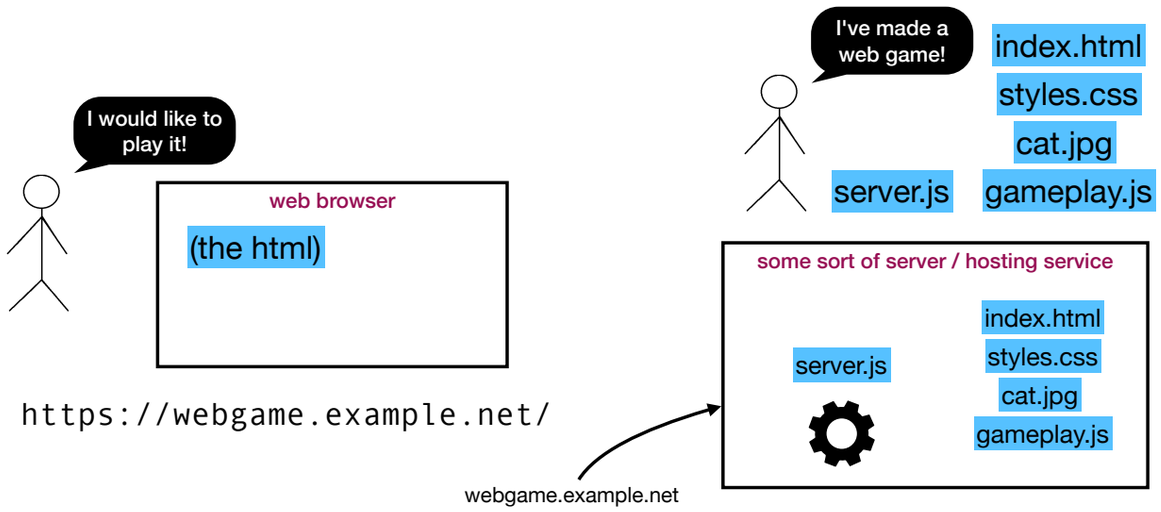
How can I play your game?



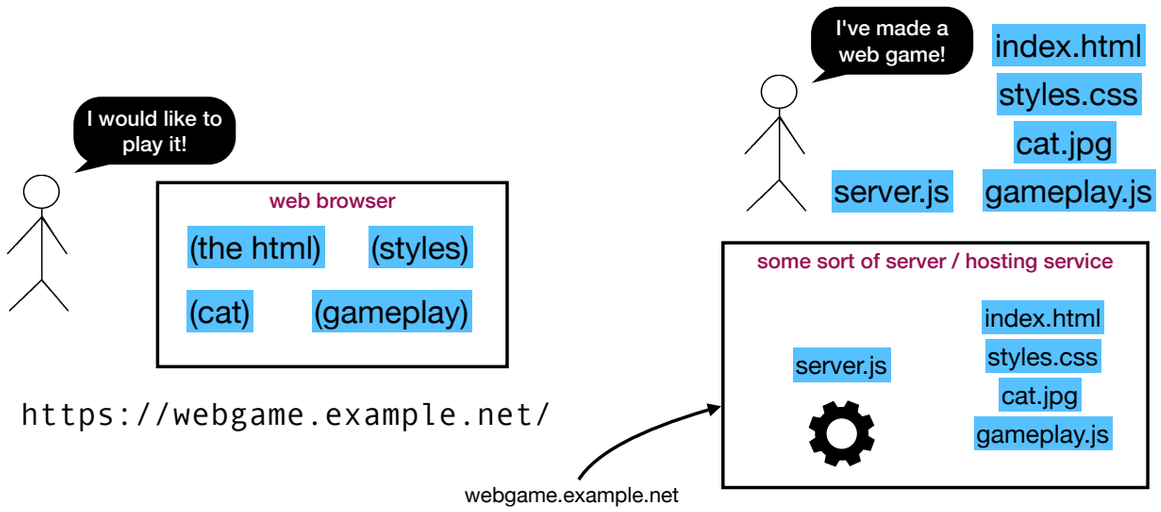
How can I play your game?



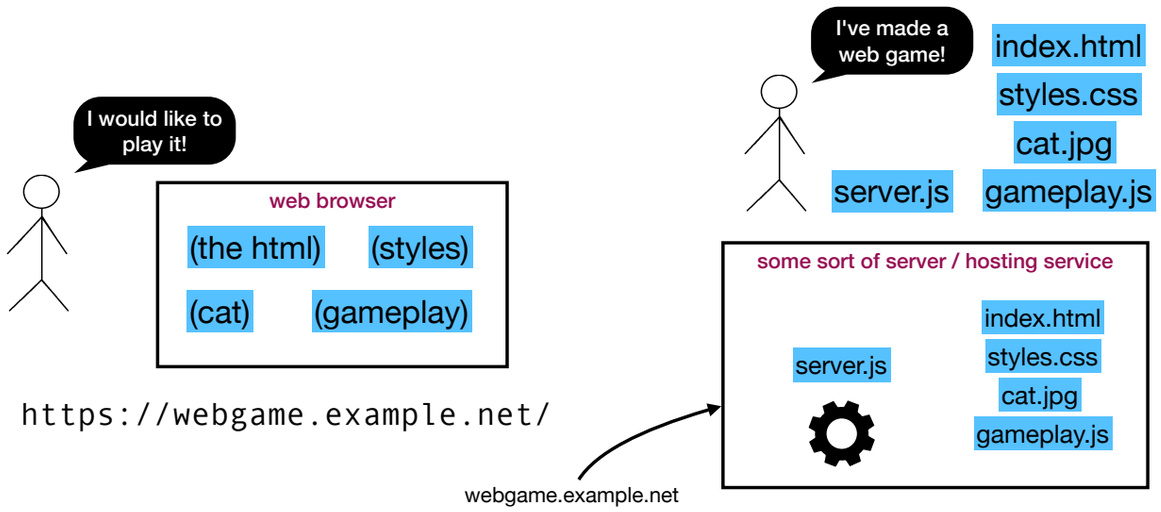
How can I play your game?



How can I play your game?

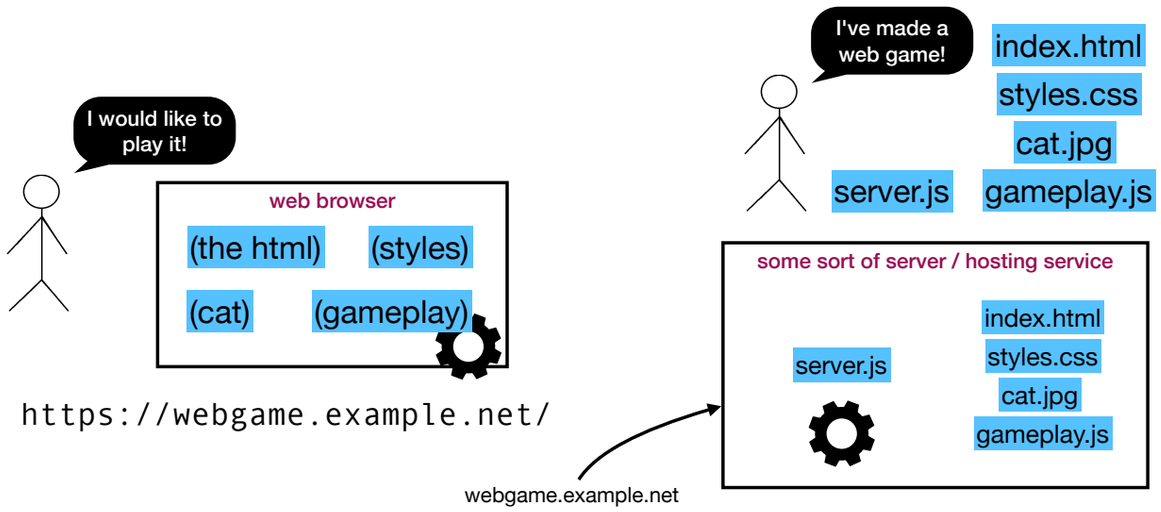


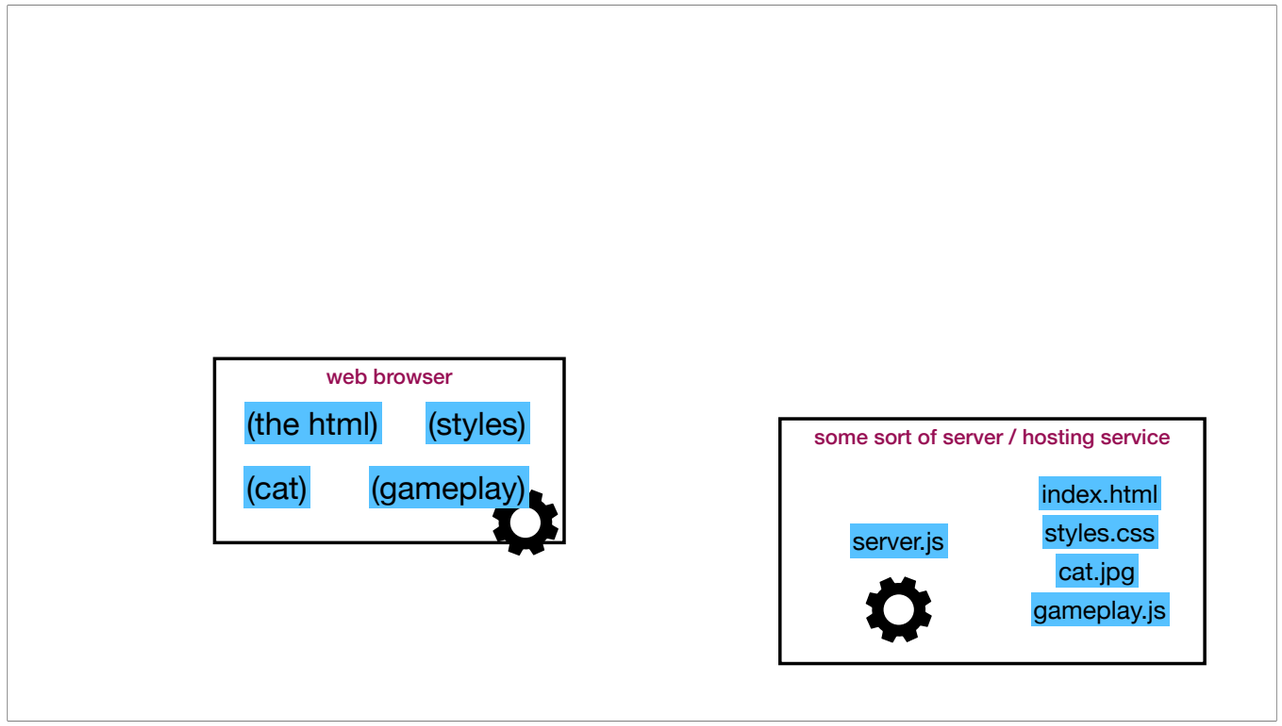
How can I play your game?

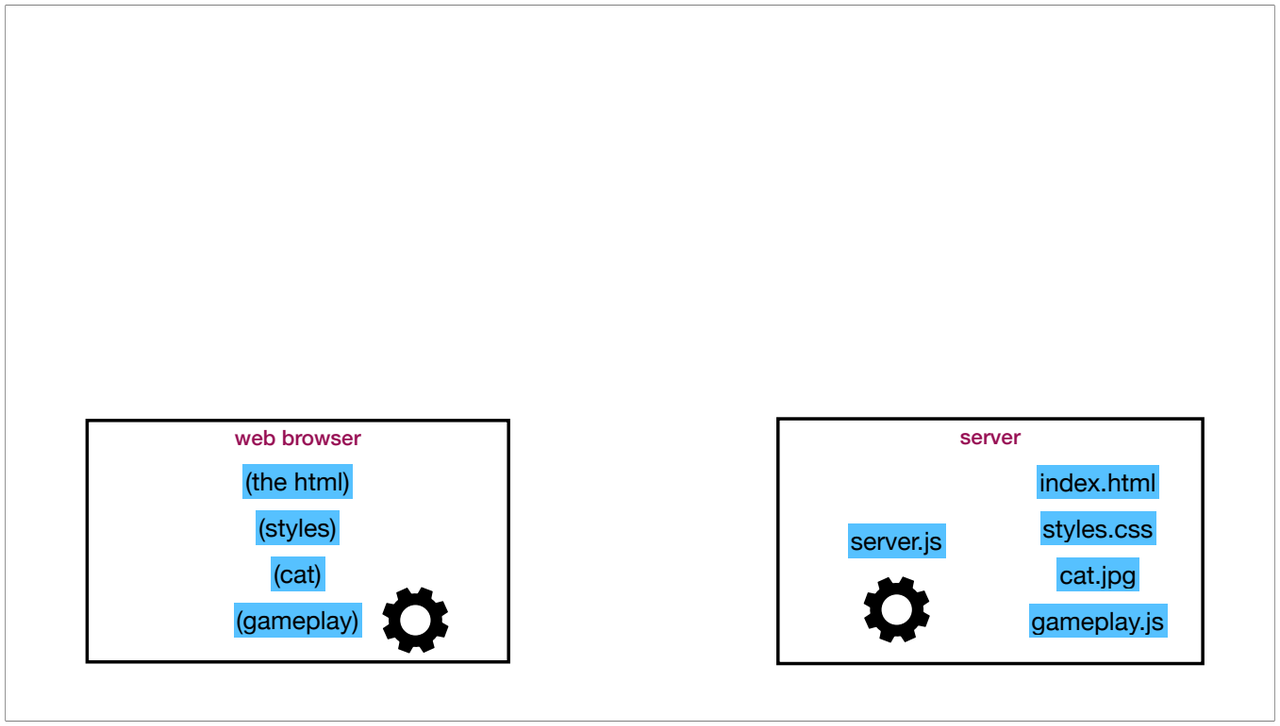


The gameplay js runs in the player's browser (and not on the server).

How can I play your game?



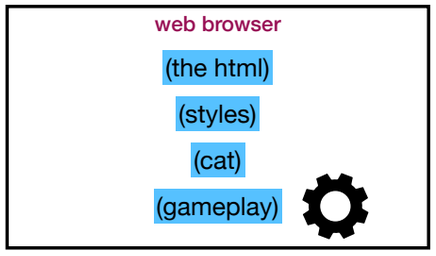

web browser
(the html)
(styles)
(cat)
(gameplay)
some sort of server / hosting service
server.js
index.html
styles.css
cat.jpg
gameplay.js


web browser
(the html)
(styles)
(cat)
(gameplay)
server
server.js
index.html
styles.css
cat.jpg
gameplay.js

Front end, Back end

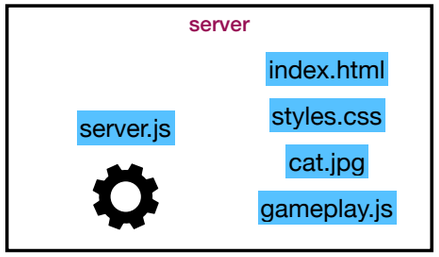
Front End

- the only programming language available is JavaScript
- the JS code can manipulate the page, fetch data, etc



Back End

- lots of choice of programming language (but we'll be using JavaScript)
- the JS code can load and save files, talk to other servers, etc



Seeing the whole URL, in your browser

i.e. https://foo.example.net/bar?aerg#arg

and not just "foo.example.net"

or even worse, "example.net"

Stuff that happens in web browsers

front-end development

Modern web browsers are complex, powerful pieces of software. As web developers, a lot of what we do is create things that are going to work — going to *run* — in web browsers.

It (usually) starts with HTML

The "starting point" request — i.e. for the URL shown in the browser address bar —is *usually* going to give a "text/html" response.

The HTML then refers to other resources, and the browser makes separate requests for them as necessary.

How HTML can refer to other resources

Just a few examples

- <img src=...>
- <script src=...>
- <link rel="stylesheet" href=...>

JavaScript in web pages is security-limited

- ✅ Read what the user types **into this page**
- ✅ React to keypresses, mouse clicks, mouse movement **in this page**
- ✅ Modify **this web page**
- ✅ Send data to / receive data from **this site's** web server

JavaScript in web pages is security-limited

- ❌ Send e-mail (or do stuff in Slack, Discord, Signal, ...)
- ❌ Read or write files on this computer (i.e. the one running the web browser)
- ❌ Make network connections (other than to this site's web server)

Using JavaScript in web pages

Typical uses

- Do something in response to buttons being clicked, things being expanded, etc
- Load more data (e.g. infinite scroll)
- Manipulate the page (show the fetched data; show a spinner; etc)

Seeing what requests your browser makes

Developer tools, Network

Stuff that happens on servers

back-end development

How the server works

It's just software. What's it doing?

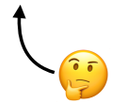
- It *listens* for connections on some predetermined port
- When it receives and accepts a connection, it talks HTTP
  - Which means, it waits for the *client* to send an *HTTP request*
  - and then it sends back the appropriate *HTTP response*

So, how does that last bit work?

How the server works

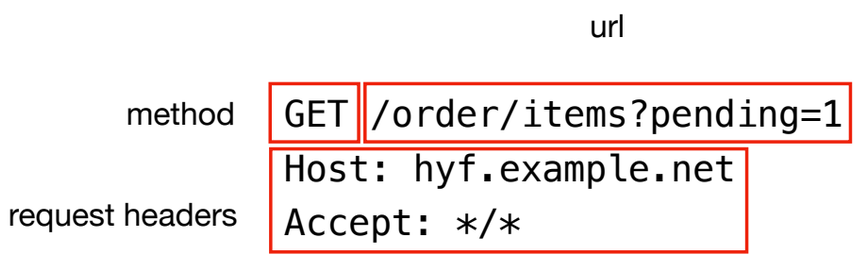
It's just software. What's it doing?

- It *listens* for connections on some predetermined port
- When it receives and accepts a connection, it talks HTTP
  - Which means, it waits for the *client* to send an *HTTP request*
  - and then it sends back the appropriate *HTTP response*



What does an HTTP request "look like"?

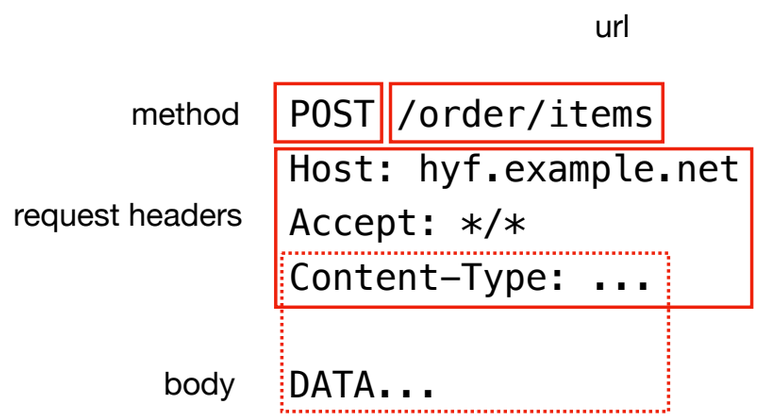
GET



i.e. as the server, our input is the request method (GET, HEAD, POST, PUT, ...), the URL, the request headers, and perhaps a content-type + body data.

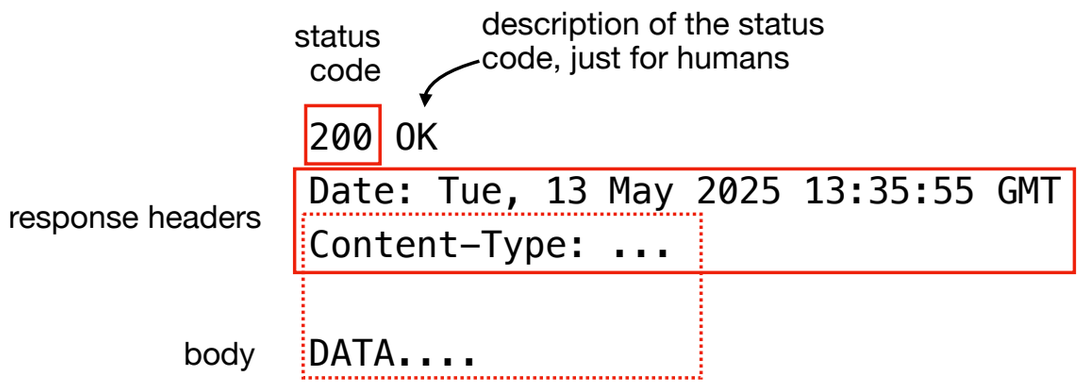
What does an HTTP request "look like"?

POST



i.e. as the server, our input is the request method (GET, HEAD, POST, PUT, ...), the URL, the request headers, and perhaps a content-type + body data.

What does an HTTP response "look like"?



i.e. as the server, we have to decide on a status code, the headers, and perhaps a content-type + body data.

Status message is the odd one out here, as it doesn't really do anything. It's for humans, not for machines.

$$f(request) \rightarrow response$$

Given a request, we have to work out what the response should be.

(But it's not purely functional. We're allowed to read / write databases, for example).

A perfectly valid web server

- for any request at all
  - reply "403 Forbidden"

Not a very *useful* server, though.

Another perfectly valid web server

- for any request at all
  - reply "404 Not Found"

Not useful, but a good default.

Static Content

which is to say: "it's just a file". For example:

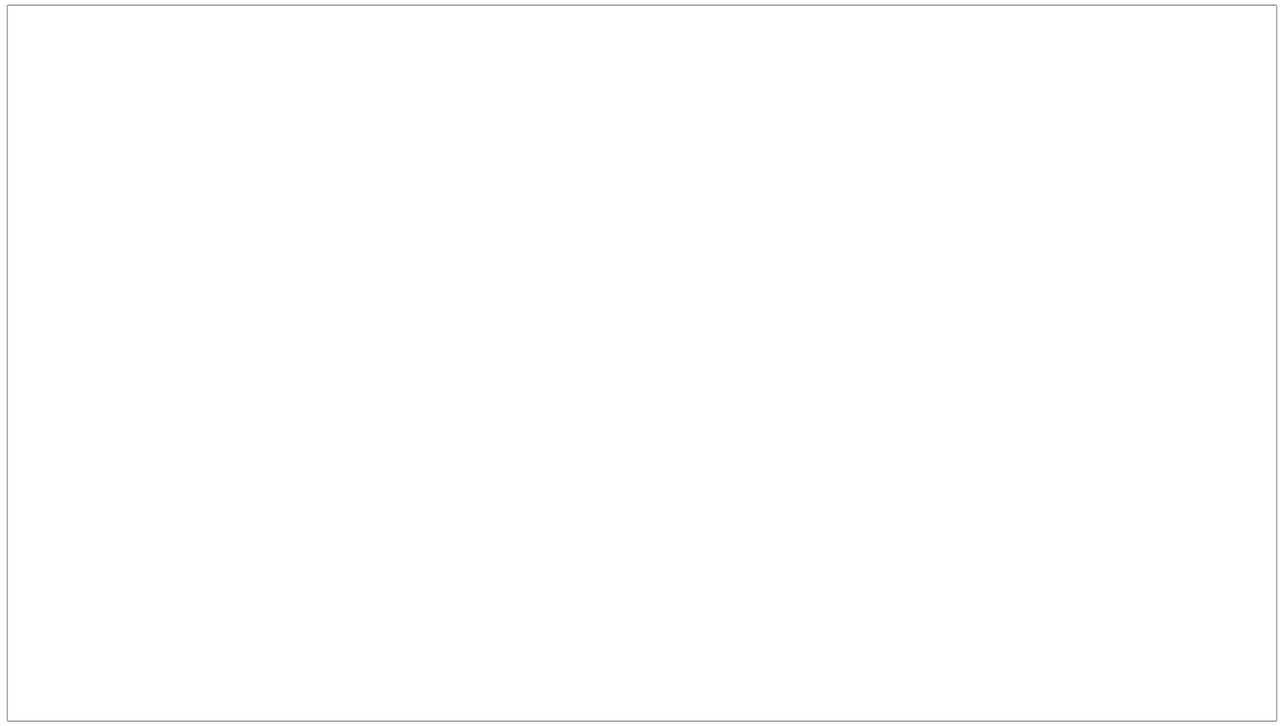
- for any request matching "GET /static/....."
  - look for a file "/home/web/project/static/....."
  - if such a file exists, then reply "200 OK" with an appropriate Content-Type and then the contents of that file
    - if it's *.html, then use Content-Type: "text/html"
if it's *.css, then use Content-Type: "text/css"
if it's *.jpg, then use Content-Type: "image/jpeg"
etc
  - otherwise (i.e. if no such file exists), reply "404 Not Found"

Because this is such a common requirement, this functionality is provided by library code, so we only have to write 1 line of code.

Commonly-used Content Types

Content-Type	Typical file extension	Description
image/jpeg	*.jpg, *.jpeg	Joint Photographic Experts Group Photographic images
image/png	*.png	Portable Network Graphics Bitmap images, e.g. screenshots
text/plain	*.txt	Just plain old text, no formatting
text/html	*.html	Hypertext Markup Language The structure and text of web pages
text/css	*.css	Cascading Style Sheets Used to control the layout and appearance of HTML
application/javascript	*.js	JavaScript The programming language of web pages

You don't need to memorise all of this right now. Just have a basic understanding of what Content Types are: a tiny piece of text (e.g. "text/html") which specifies what format some bit of data (the "content") is in.



We'll come back to more complex server behaviour later.

TODO, add some back-end-y kind of exercises here.

Just show me the data

[a brief intro to JSON, APIs, and why]

Let's use an API!

(API = "Application Programming Interface", but no-one says that)

Scenario: a comments section.

- a button to check for new comments
- a form to add a new comment

What does the data look like?

- a comments section has an ordered list of saved comments
- a comment is a timestamp, the comment text, and the (self-reported) name of who said it

{ "timestamp": "...", "text": "...", "author": "..." }

HTTP methods

- GET
- HEAD

- POST

and more. But those are the most common ones.

Idempotency (in simple terms).

GET allows for query parameters.
POST allows content (of some content-type) to be sent.

What would the HTTP requests look like?

New comment:
POST /api/comments (with text and author)
server adds the timestamp, and adds to the list of comments.
replies OK.

Fetch all comments:
GET /api/comments
(an array of comments comes back in the response body)

Front end: Using an API

- on "check for new comments" button click:
  - issue a GET /api/comments
  - and when the results come back, update the page accordingly

- new comment form:
  - two fields: text, and author
  - on submit, issue a POST /api/comments with the data
  - when the OK comes back, clear the form, and do the same as "check for new comments"

Back end: Providing an API

- if method is POST and url is "/api/comments"
  - check that we got given "text" and "author"
  - add timestamp (current time)
  - add the new comment to wherever we store our comments

- if method is GET and url is "/api/comments"
  - fetch all comments, from wherever we store our comments
  - send them back in the response

TODO, add some API-y exercises here

Going beyond a single server

Just an introduction to the types of things which are possible.

Let's add a database

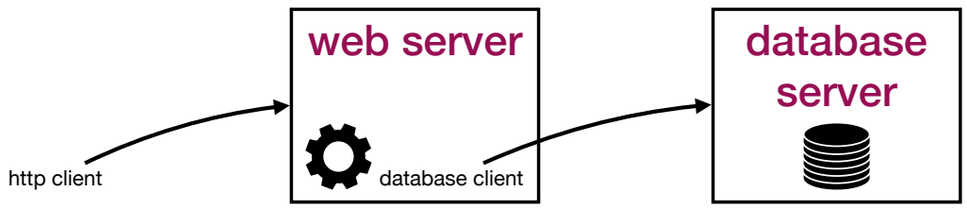
web + postgres / mysql

Who can you trust?

gear = our software.

So: can we trust the requests which arrive at our web server?

Who can you trust?



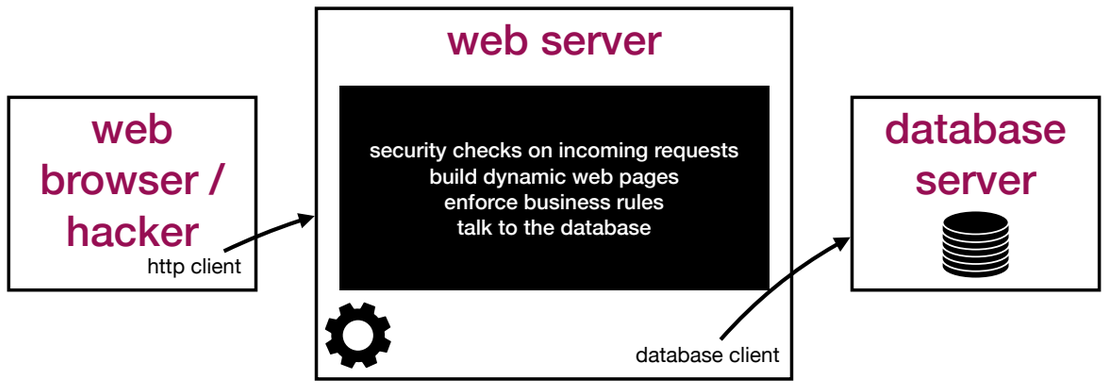
all we really know for sure is that it's coming from something which speaks HTTP...

Who can you trust?

Uh oh! There's a bad person on the Internet!

So we can't trust requests that come in to the server. We have to be careful how we process the requests.

What does the server need to do?

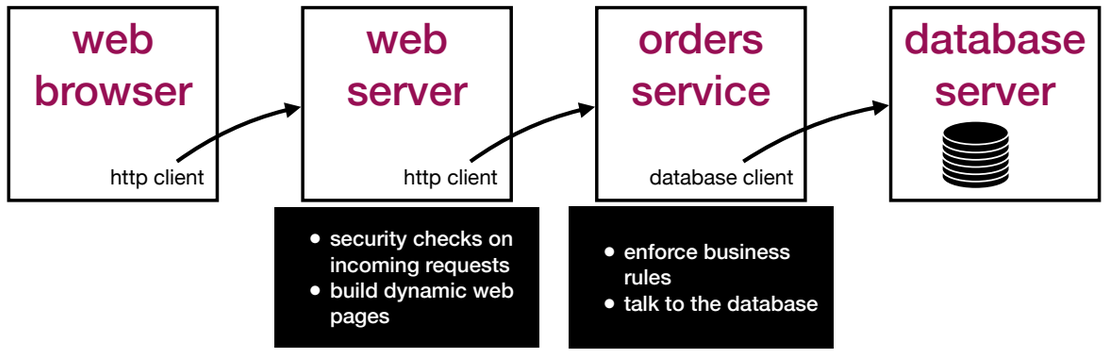


Uh oh! There's a bad person on the Internet!

So we can't trust requests that come in to the server. We have to be careful how we process the requests.

What else does the server need to do?

Let's add an orders service!



Going even bigger

Multiple environments

Multiple regions / points of presence

> 1 env

Recap

Front end vs back end

Web development spans:

- the **front end** (stuff that happens in the users' web browsers)
- the **back end** (stuff that happens on servers that we control).

The front end and the back end communicate via **HTTP**. The front end makes a request, and the back end answers with a response.

Recap

What happens in the front end

We get to write the software on both sides:

- in the front end:
  - we can *only* use JavaScript
  - we can do things like manipulate the web page, or make a request to the server

Recap

What happens in the back end

We get to write the software on both sides:

- in the back end:
  - the primary goal is to send back a response for each HTTP request
  - we can use any language we like (but we'll be using JavaScript)
  - we can also do things like read and write files (on the server), or talk to a database, or to another server

Recap

Deployment

The server is ours to control. It runs the back end code.

It also has a *copy of* the front end code, so that it can provide it to the front end, for the web browser to run it.

"Deployment" is about making sure that the server has all the code + other files and settings that it needs.

the end

More HTTP response (status) codes

Just the most commonly-seen ones

- 200 OK
- 301/302/303/307/308 — various forms of redirection
- 404 Not Found
- 403 Forbidden
- 500 Server Error