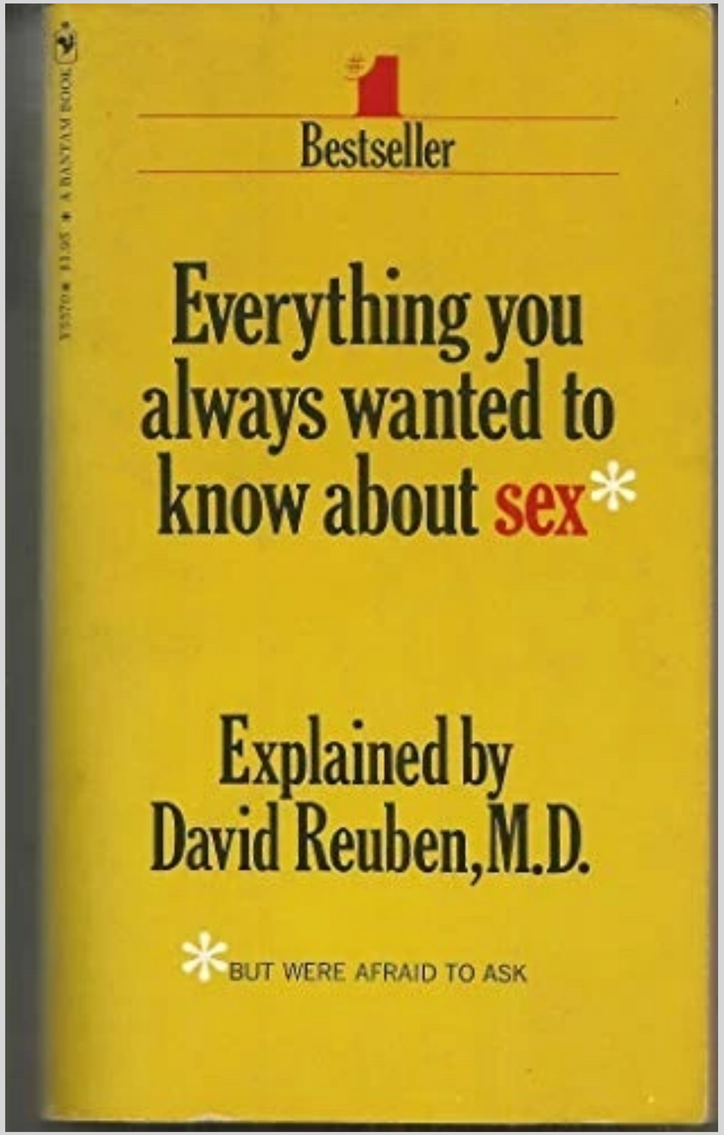
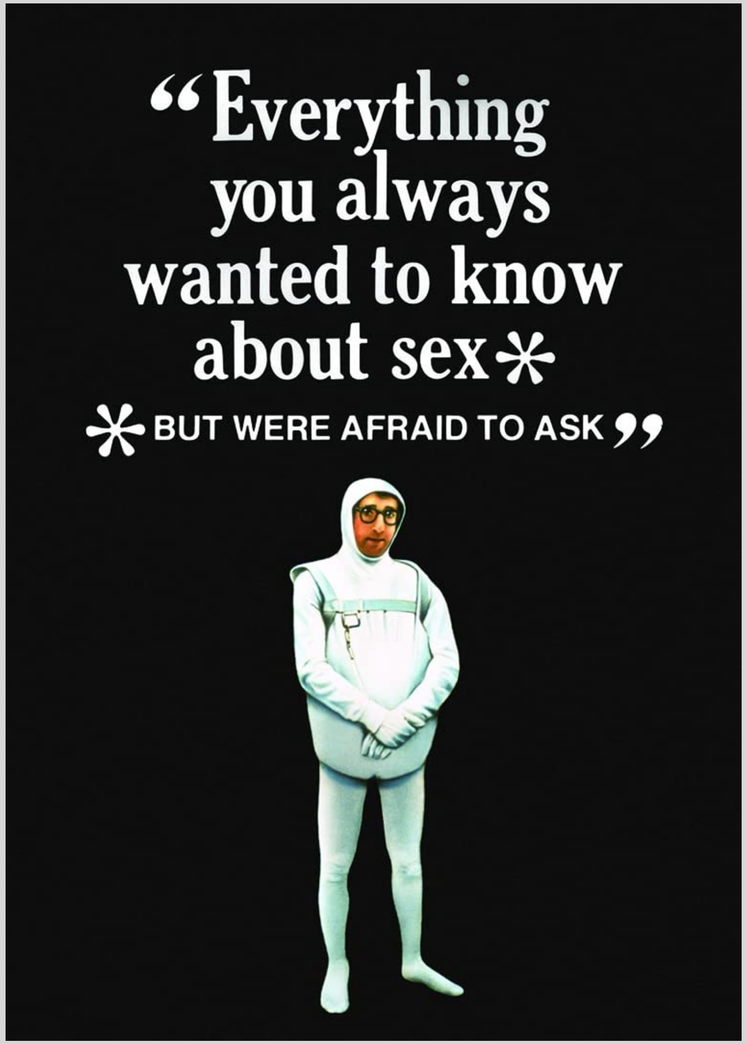
everything you always wanted to know about SE* but were afraid to ask

(*software engineering of web apps)

Daniel Jackson



bestselling book
(David Reuben, 1969)



satirical movie
(Woody Allen, 1972)

what you'll learn today

how web apps work

the idea of client-server computing

what's client-server?

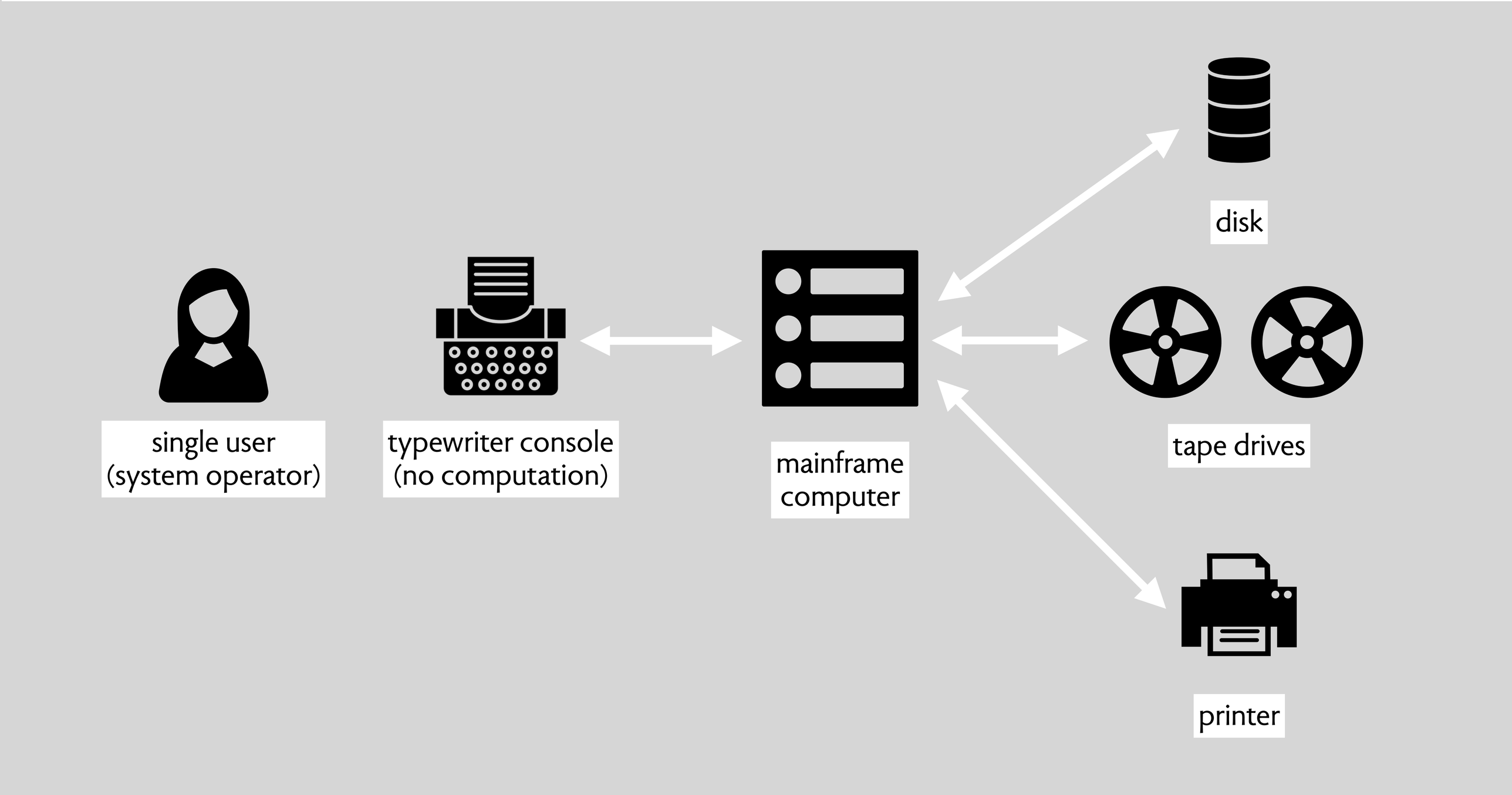
mainframe computer



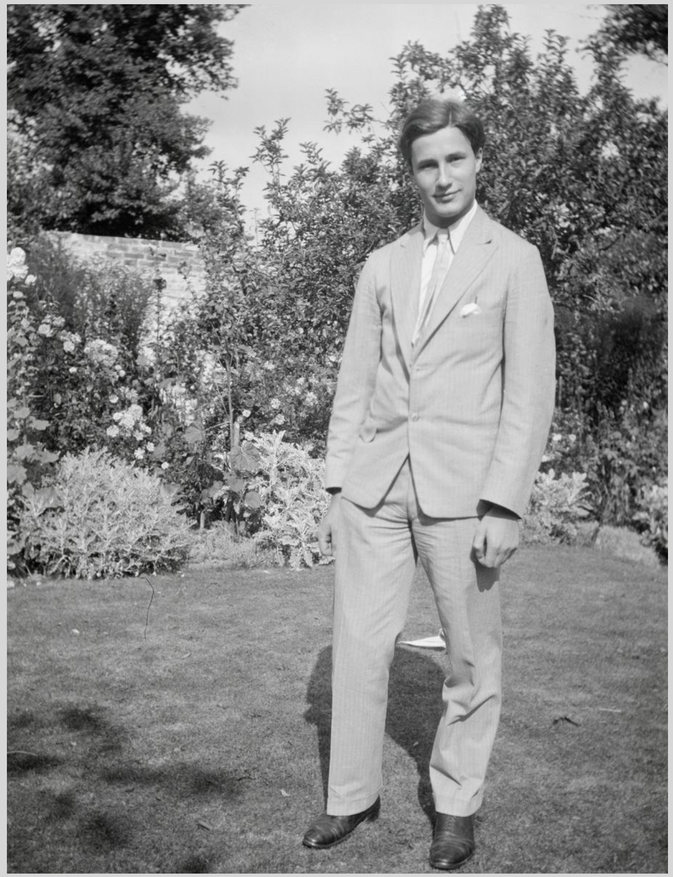
no real client
console is wired in part of machine

Univac 1 at Census Bureau: first commercial computer for civilian government agency (1951)

early mainframe architecture (1962)



the invention of time sharing



Christopher Strachey in 1936
patented time sharing (1959)
denotational semantics (1966)



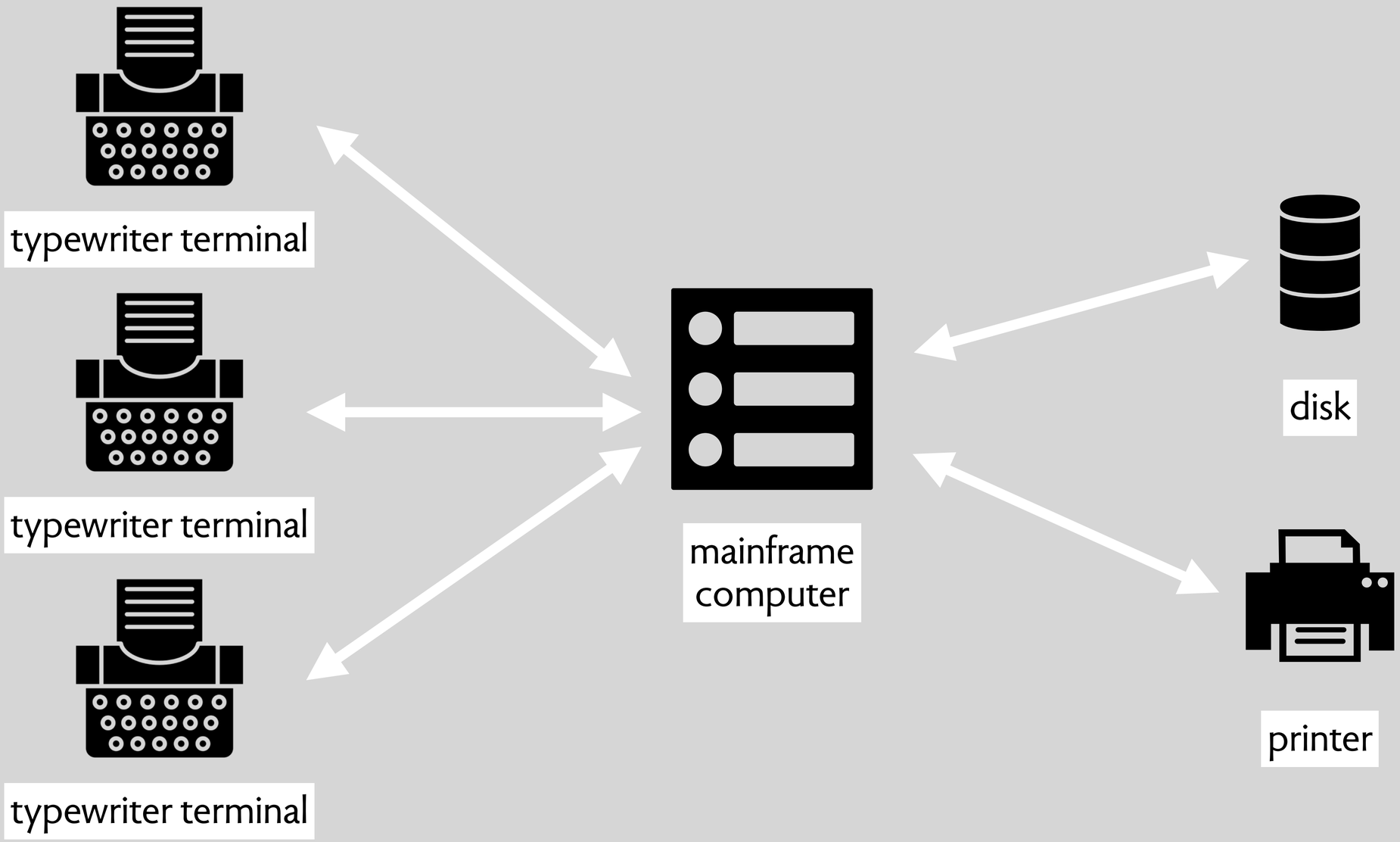
my father's high school teacher
at Harrow School in London
wrote checkers program (1951)
with help from Alan Turing

timesharing at MIT: Project Mac and CTSS



Fernando J. Corbató with IBM 7090 (1962)

time sharing architecture (early 1960s)



first personal computer



Altair 8800 (1974)
hobbyist kit, required assembly



Paul Allen and Bill Gates (1975)
wrote BASIC interpreter for Altair

first mass-market personal computers



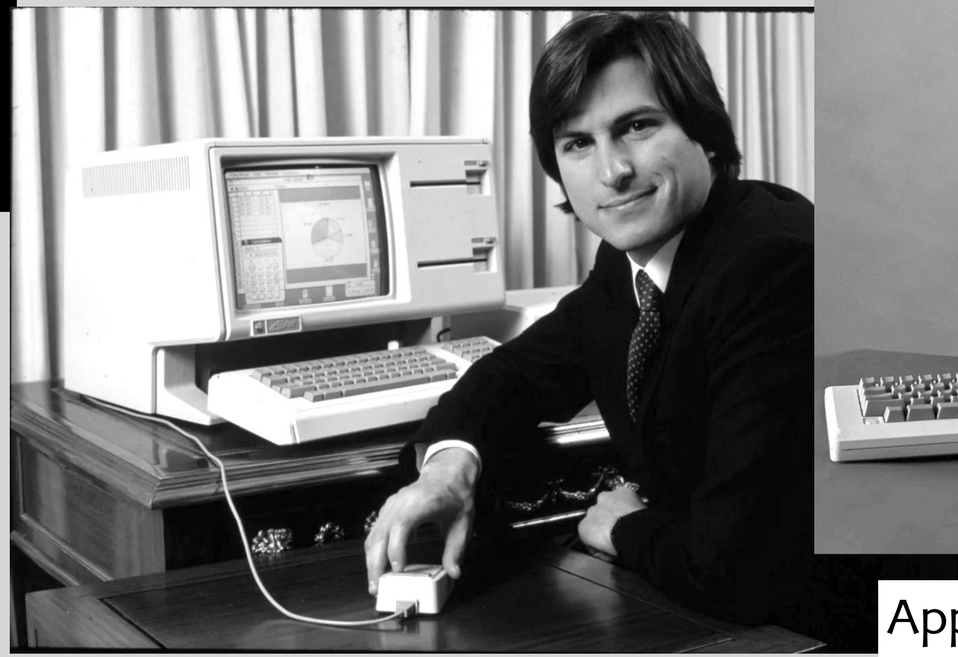
Commodore PET 2001 (1977)
all in one computer
for schools, home users, businesses

IBM PC (1981)

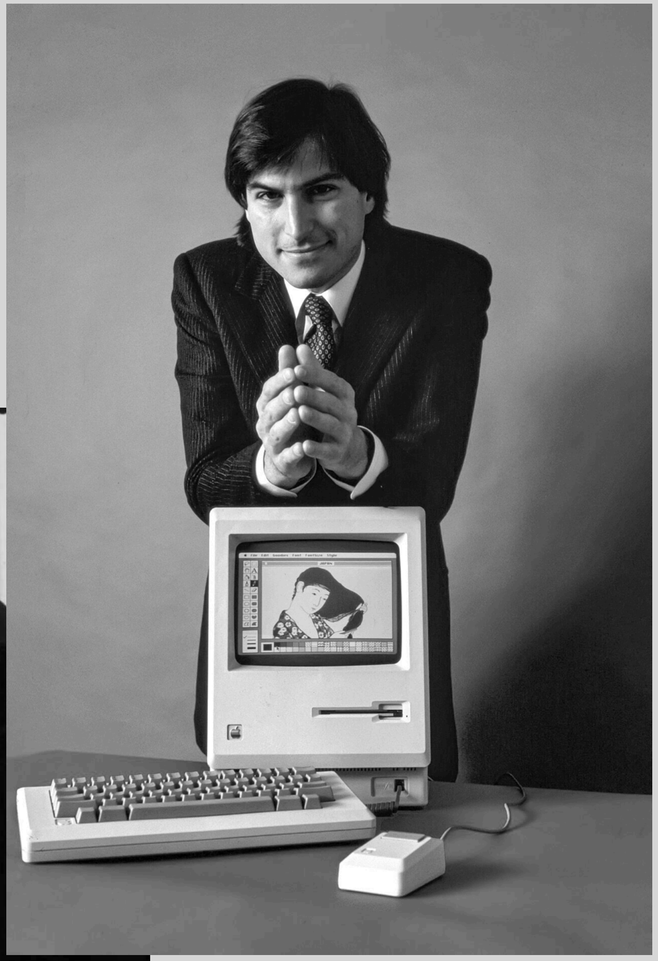




Apple 1 (Steve Jobs and Steve Wozniak, 1976)
preassembled motherboard: bring your own box



Apple Lisa (1983)
First consumer computer
with mouse & GUI
cost $10k



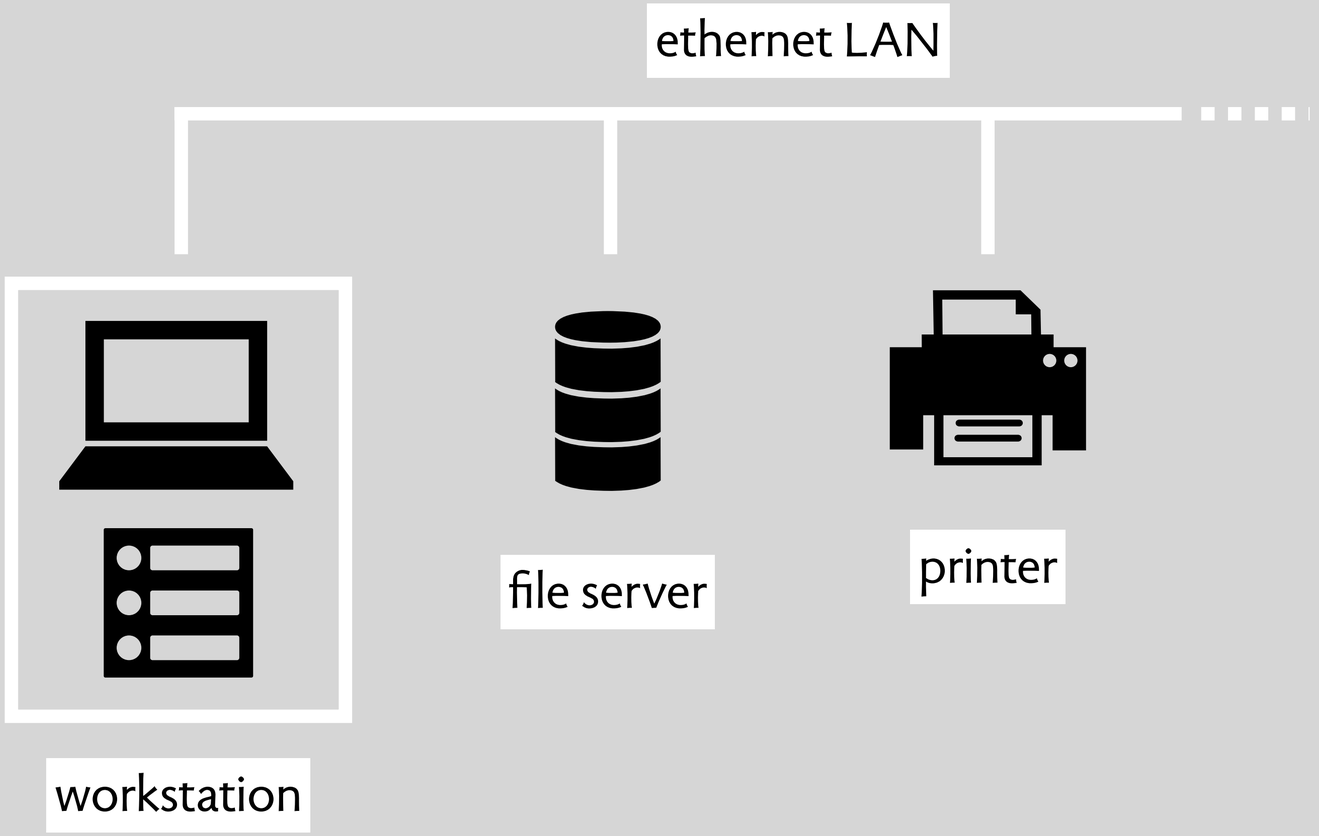
Apple Macintosh (1984)
cost $2500

what was missing?
that your laptop
has today

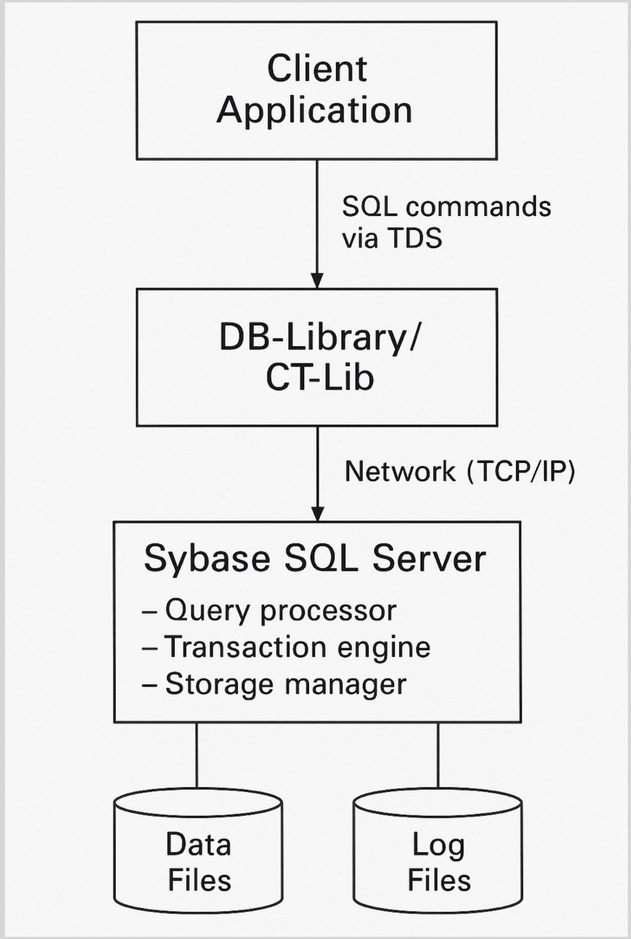
first client-server systems (1970s)



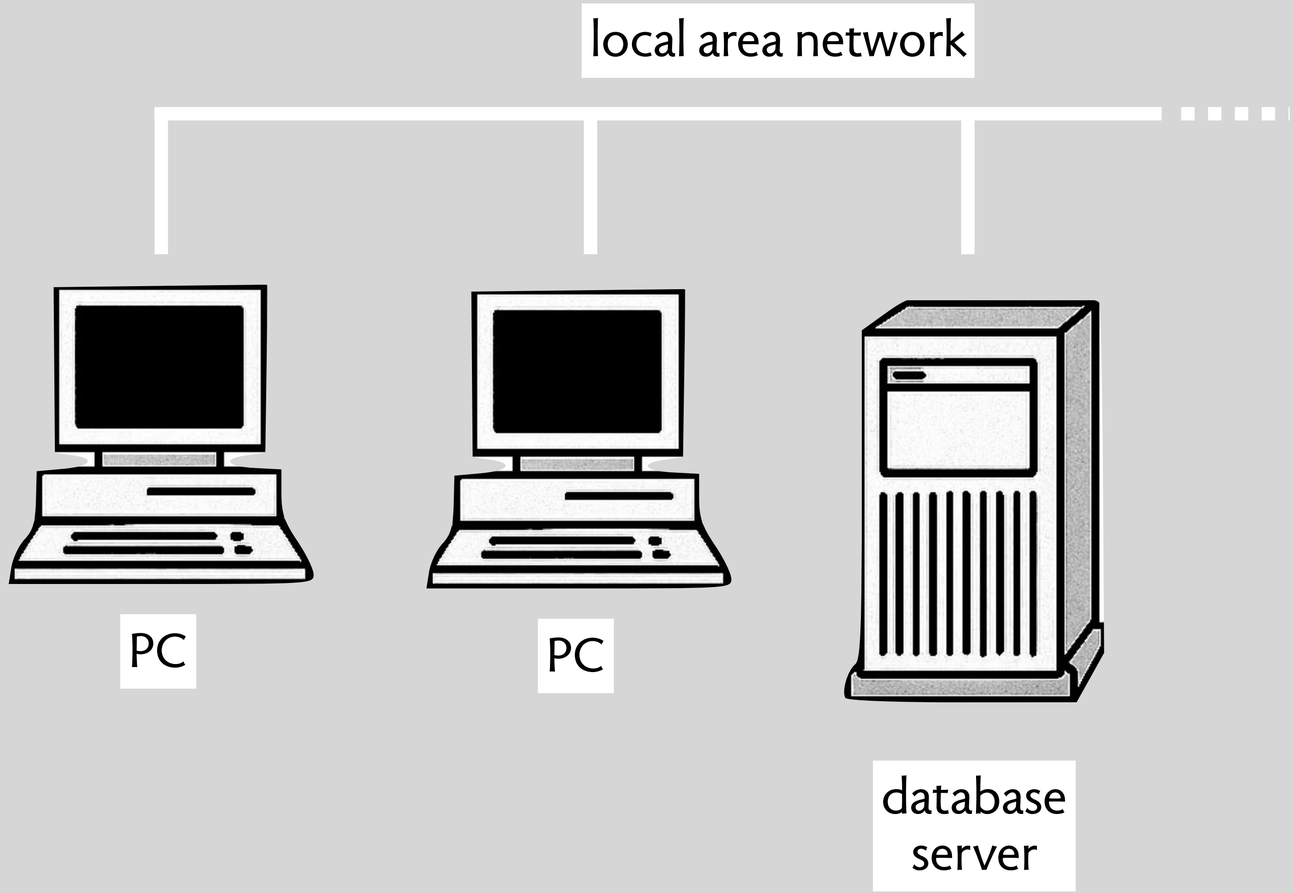
Xerox Alto (1973)
first "client-server" system
client is not just a terminal



first client-server database systems (1980s)



Sybase SQL Server (1986)
apps running on PCs
SQL queries to database server
no internet yet



what's HTTP?

TBL's proposal (1989)

Vague but exciting ...

CERN DD/OC

Tim Berners-Lee, CERN/DD

Information Management: A Proposal

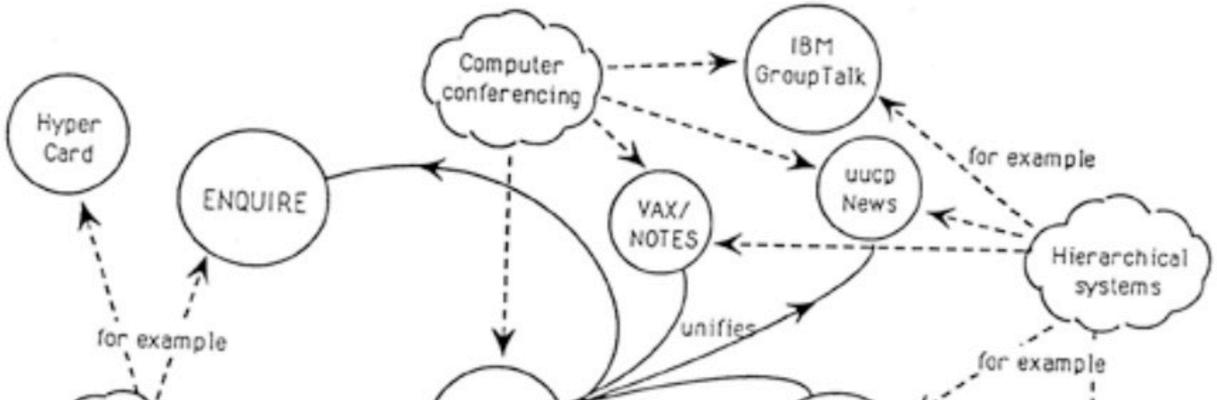
March 1989

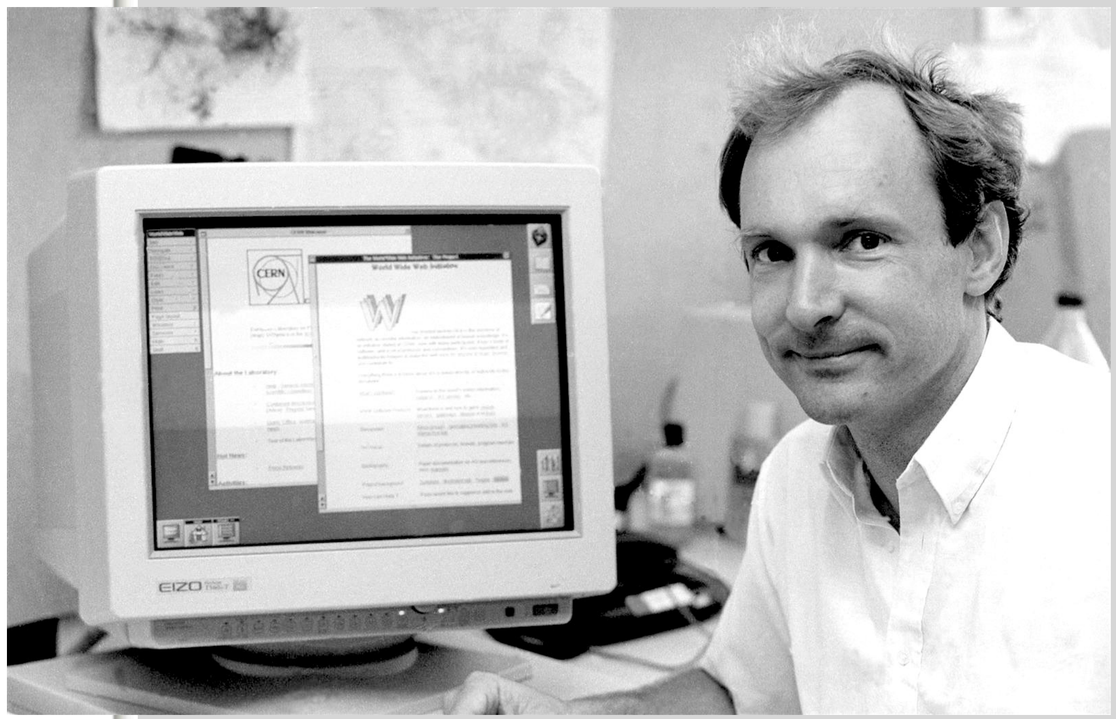
Information Management: A Proposal

Abstract

This proposal concerns the management of general information about accelerators and experiments at CERN. It discusses the problems of loss of information about complex evolving systems and derives a solution based on a distributed hypertext sytstem.

Keywords: Hypertext, Computer conferencing, Document retrieval, Information management, Project control





what was different about the web from previous client-server systems?



GU4P9R

https://class.mit-sdg.dev/join

the first web page (1991)

World Wide Web

The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an executive summary of the project, Mailing lists , Policy , November's W3 news , Frequently Asked Questions .

What's out there?
: Pointers to the world's online information, subjects , W3 servers, etc.

Help
: on the browser you are using

Software Products
: A list of W3 project components and their current state. (e.g. Line Mode ,X11 Viola , NeXTStep , Servers , Tools , Mail robot , Library)

Technical
: Details of protocols, formats, program internals etc

Bibliography
: Paper documentation on W3 and references.

People
: A list of some people involved in the project.

History
: A summary of the history of the project.

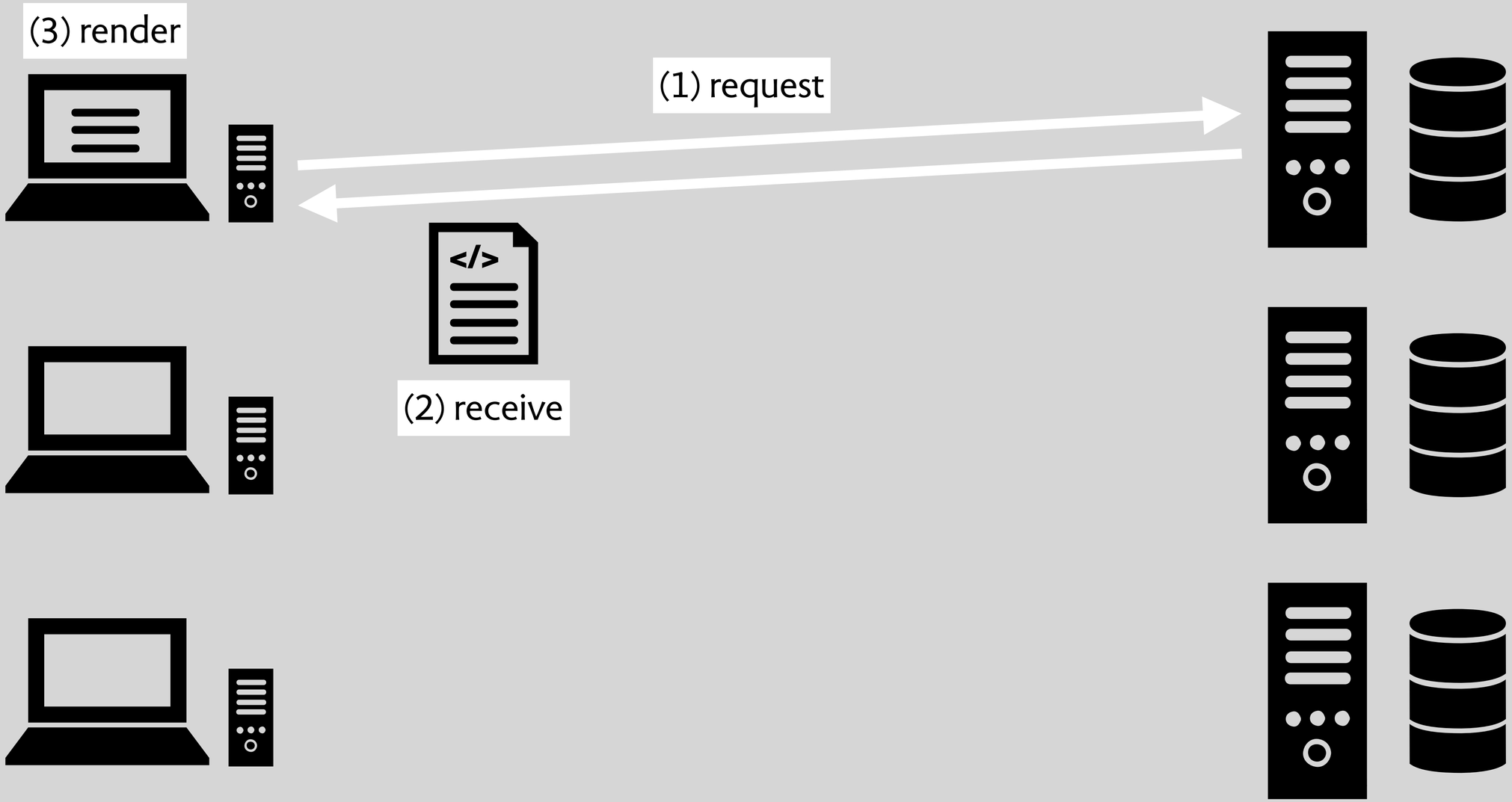
How can I help ?
: If you would like to support the web..

Getting code
: Getting the code by anonymous FTP , etc.

essential ingredients

URLs + HTML + HTTP

TBL's web (1991)



the web as a document editing system: HTTP verbs

Create	→	**POST**
Read	→	**GET**
Update	→	**PUT**
Delete	→	**DELETE**

HTTP verbs
many originally proposed (1992)
eventually settles on these

distributed document editing
TBL's original concept of the web

did this actually happen?

structure of a URL

```
https://api.example.com/users/123/profile?view=full&include=posts&lang=en
```

Component	Example	Description
Scheme / Protocol	`https://`	Specifies secure HTTP (HTTPS)
Domain name	`api.example.com`	The server's hostname
Path	`/users/123/profile`	The route to a specific resource
Query parameters	`?view=full&include=posts&lang=en`	Key–value pairs providing extra info for the request
Full meaning	"Request the full profile of user 123 from the API, including their posts, in English."	

elements of an HTTP request

Element	Example	Description
Method / Verb	`GET`, `POST`, `PUT`, `PATCH`, `DELETE`	Specifies the action the client wants the server to perform on the resource.
Path	`/api/users/42?verbose=true`	The resource path and optional query parameters — relative to the host. (In most requests, this replaces the full URL.)
HTTP Version	`HTTP/1.1`	Indicates the HTTP protocol version being used.
Headers	`Content-Type: application/json` `Authorization: Bearer abc123`	Key–value pairs carrying metadata about the request: content type, authentication, caching, cookies, etc.
Host	`Host: api.example.com`	Specifies the target host (required in HTTP/1.1 and later).
Body / Payload	`{ "email": "new@example.com" }`	Optional data sent with the request — used mainly in `POST`, `PUT`, and `PATCH` methods.

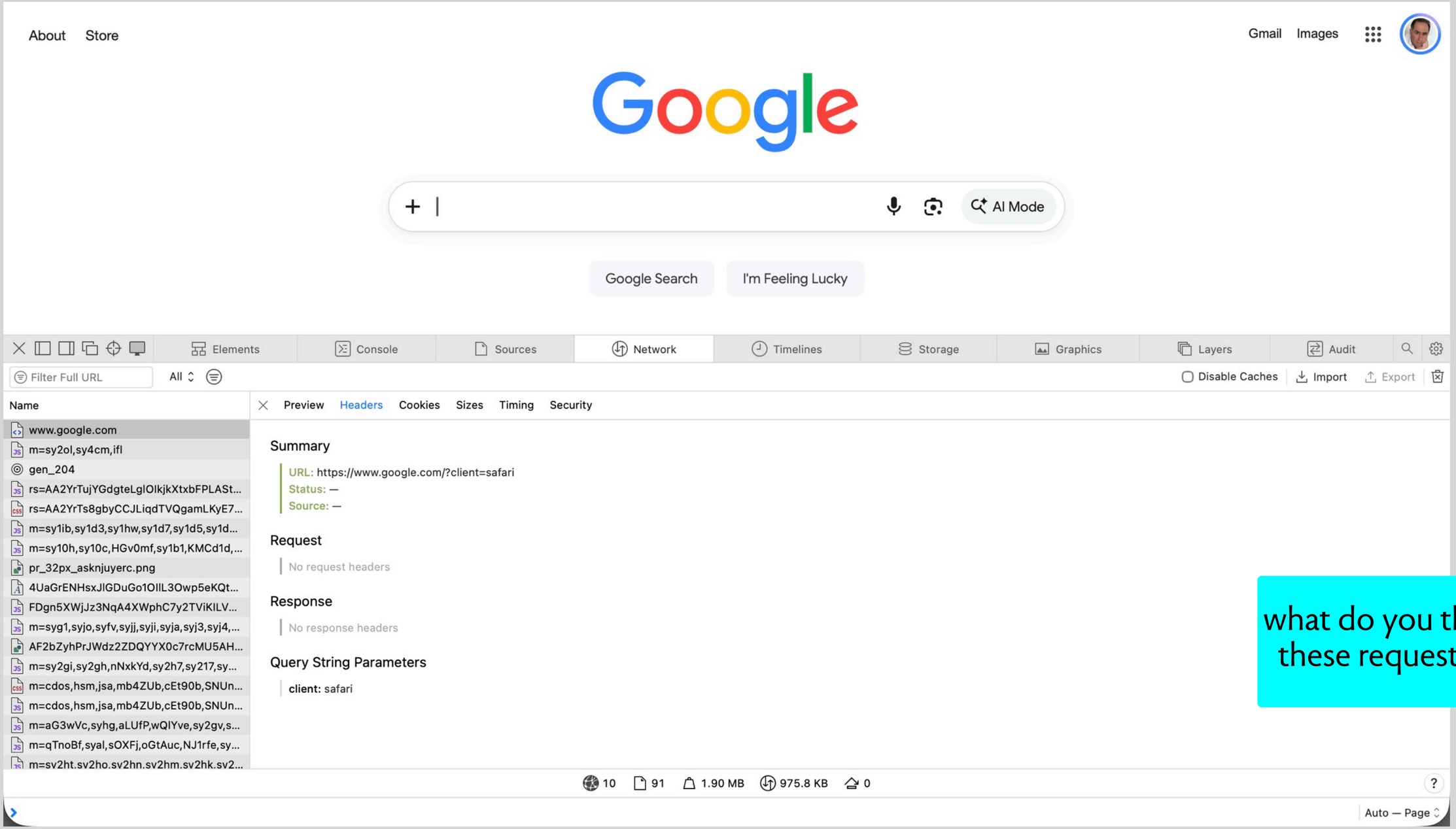
REST: a standard pattern for routes

HTTP Request Type	URL Path	Description
GET	'/articles'	display a list of all articles
GET	'/articles/new'	return an HTML form for creating a new article
POST	'/articles'	create a new article
GET	'/articles/:id'	display a specific article
GET	'/articles/:id/edit'	return an HTML form for editing a article
PUT	'/articles/:id'	update a specific article
DELETE	'/articles/:id'	delete a specific article

Representational State Transfer (Roy Fielding, 2000)
example here for Ruby on Rails style

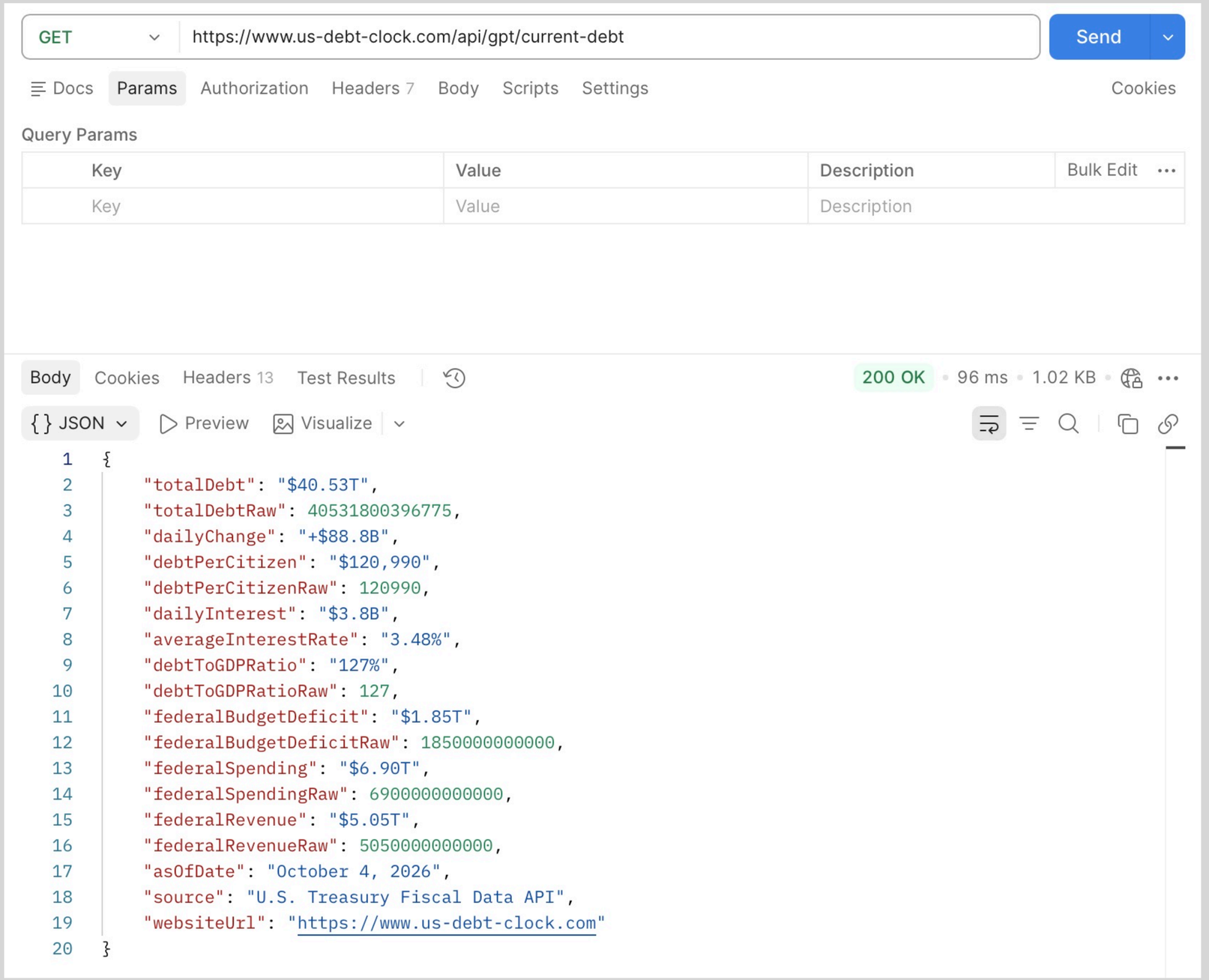
essence of the idea
nouns (URL, tree)
verbs (HTTP method)

looking at HTTP requests with dev tools



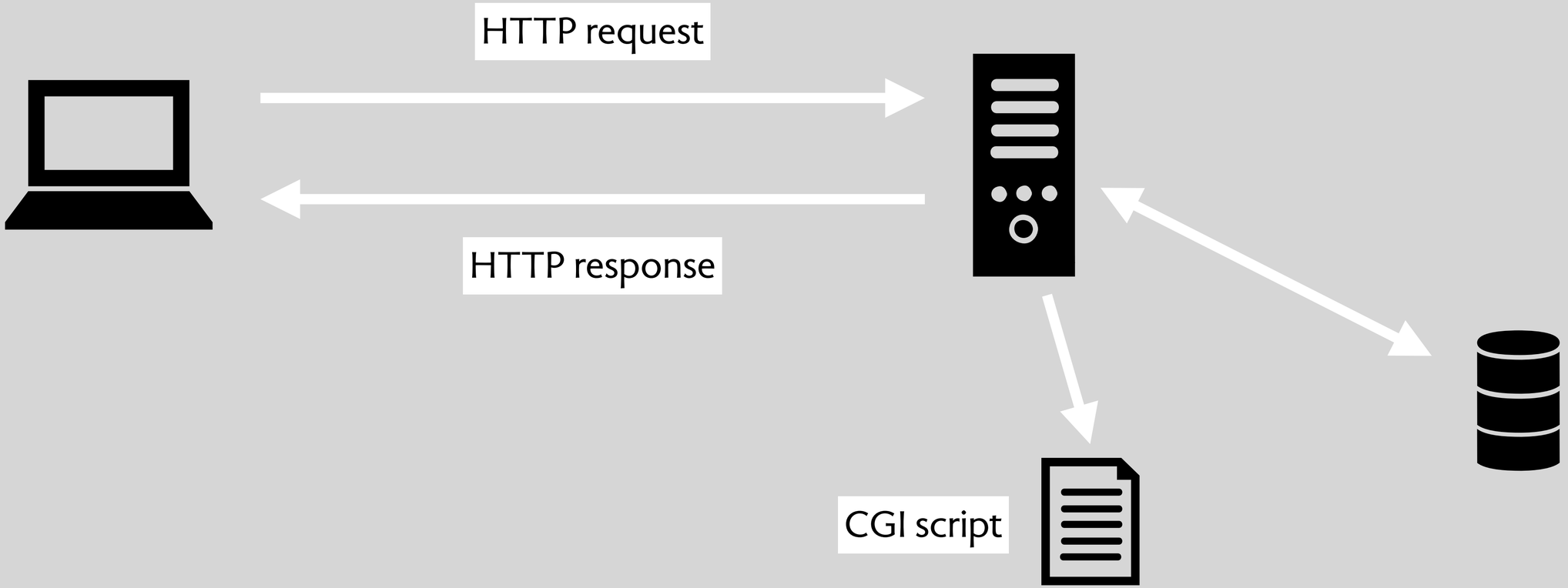
what do you think all these requests do?

issuing requests with postman



how did web sites become web apps?

common gateway interface (1993)



hello world CGI script

```
#!/usr/bin/perl
print "Content-type: text/html\n\n";
print "Hello, World.";
```

a famous CGI script

```perl
#!/usr/bin/perl

# Read form input
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
@pairs = split(/&/, $buffer);
foreach $pair (@pairs) {
    ($name, $value) = split(/=/, $pair);
    $value =~ tr/+/ /;
    $value =~ s/%(..)/pack("C", hex($1))/eg;
    $FORM{$name} = $value;
}

# Get recipient from the form field
$recipient = $FORM{'recipient'};

# Send mail
open(MAIL, "|/usr/lib/sendmail -t");
print MAIL "To: $recipient\n";
print MAIL "From: $FORM{'email'}\n";
print MAIL "Subject: Form submission\n\n";
print MAIL "Here are the results:\n";
foreach $key (keys %FORM) {
    print MAIL "$key = $FORM{$key}\n";
}
close(MAIL);
```

FormMail (1995)
contact form email
written by Matt Wright
a high school student



what's the worst that can happen if client modifies recipient?

script didn't sanitize input
client can add extra fields
so use as spam relay
by early 2000s, blacklisted

recipient=admin@example.com%0ABcc:spamlist@example.org

a CGI-powered web app (1995)



Welcome to Amazon.com Books!

One million titles,
consistently low prices.

(If you explore just one thing, make it our personal notification service. We think it's very cool! Also, check out what our customers are saying about us.)

Spotlight! -- January 16th

These are books we love, offered at Amazon.com low prices. The spotlight moves **EVERY** day so please come often.



One Million Titles

Search Amazon.com's million title catalog by author, subject, title, keyword, and more... Or take a look at the books we recommend in over 20 categories... Check out our customer reviews and the award winners from the Hugo and Nebula to the Pulitzer and Nobel... and bestsellers are 30% off the publisher's list...



Eyes & Editors, A Personal Notification Service

Like to know when that book you want comes out in paperback or when your favorite author releases a new title? Eyes, our tireless, automated search agent, will send you mail. Meanwhile, our human editors are busy previewing galleys and reading advance reviews. They can let you know when especially wonderful works are published in particular genres or subject areas. Come in, meet Eyes, and have it all explained.

Your Account

Check the status of your orders or change the email address and password you have on file with us. Please note that you

running code in the browser

```
<div id="box" style="position:absolute; left:0px; top:100px;">🌟</div>
<script>
  let x = 0;
  function move() {
    x += 5;
    document.getElementById("box").style.left = x + "px";
    if (x < 300) setTimeout(move, 30);
  }
  move();
</script>
```

JavaScript (1995) and standard DOM (1998)



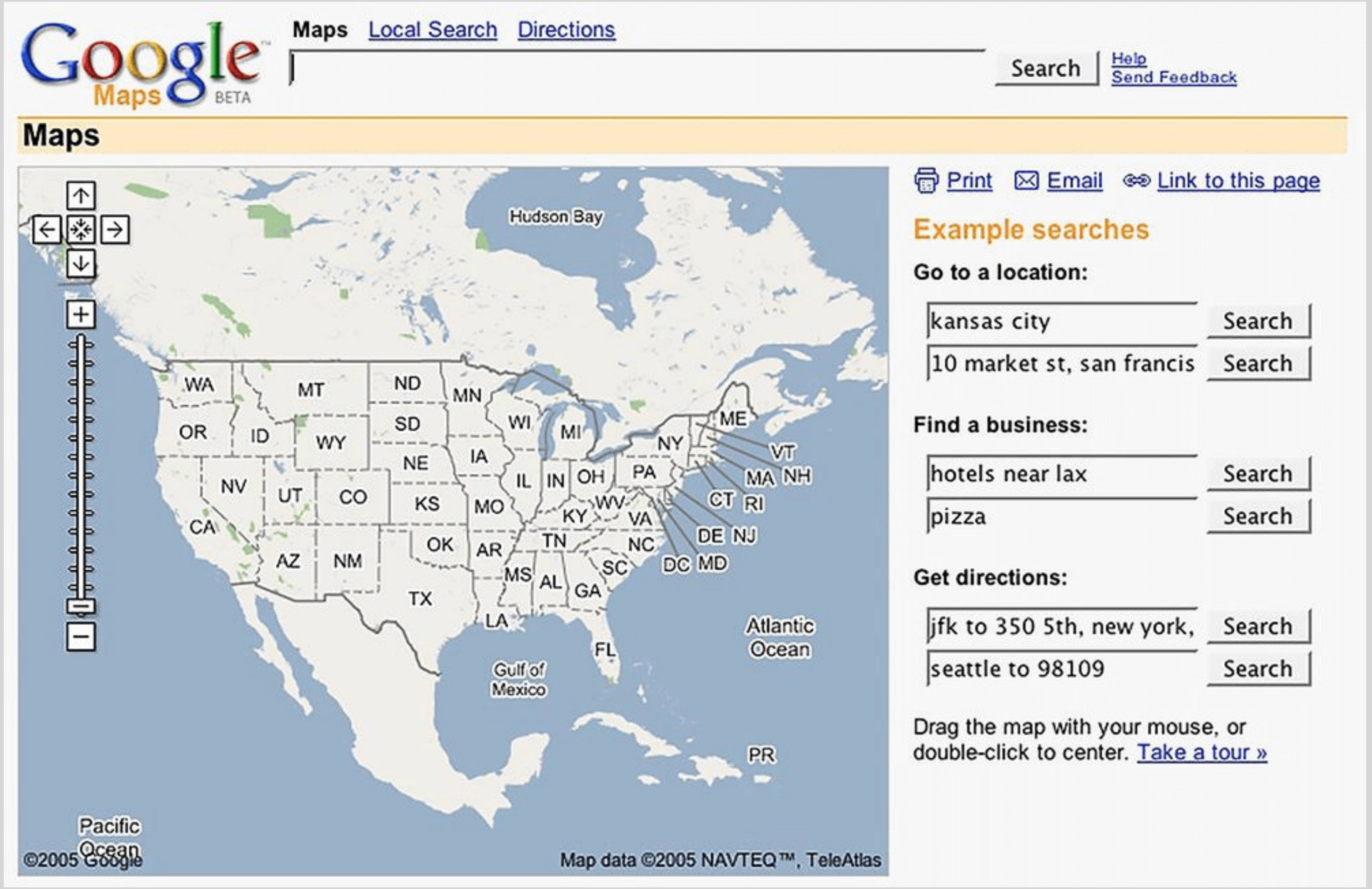
AJAX: asynchronous JavaScript and XML

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "example.txt", true);
xhr.onreadystatechange = function() {
  if (xhr.readyState === 4 && xhr.status === 200) {
    console.log(xhr.responseText);
  }
};
xhr.send();
```

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "/submit", true);
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xhr.send("name=Daniel&message=Hello");
```

XMLHttpRequest (1998)
calling server inside a script in the browser
introduced by Microsoft
later standardized for all browsers

an AJAX-powered web app (2005)



what server calls look like now

```
const api = axios.create({
  baseURL: '/api',
  headers: {
    'Content-Type': 'application/json',
    'Accept': 'application/json'
  },
  timeout: 10000
})
```

await
semantically, a blocking call
as if this “thread” waits for return
but other events still processed

```
async register(username, password) {
  const response = await api.post('/UserAuth/register', {
    username,
    password
  })
  return response.data
},
```

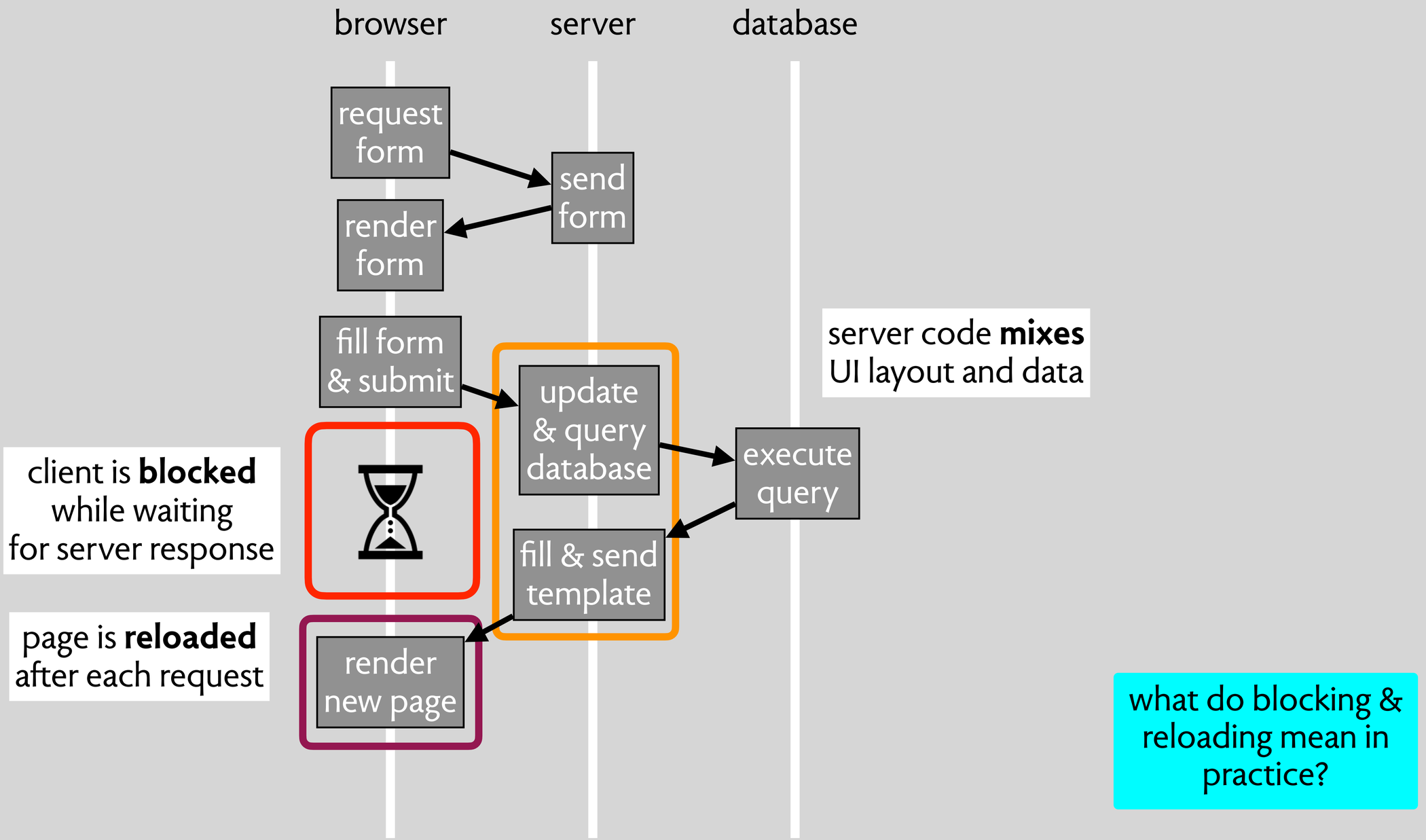
```
const result = await register(form.value.username, form.value.password)

if (result.success) {
  // Redirect to home page after successful registration
  router.push('/')
```

now like a local call
but asynchronous

what's an SPA?

the flow for a "multi-page" app



using a template to render HTML in the server

```python
@app.route("/user/<name>")
def user_profile(name):
    # Data fetched from a database, for example
    user_info = {
        "name": name,
        "age": 32,
        "hobbies": ["music", "sailing", "AI"]
    }
    return render_template(
        "profile.html",
        user=user_info
    )
```

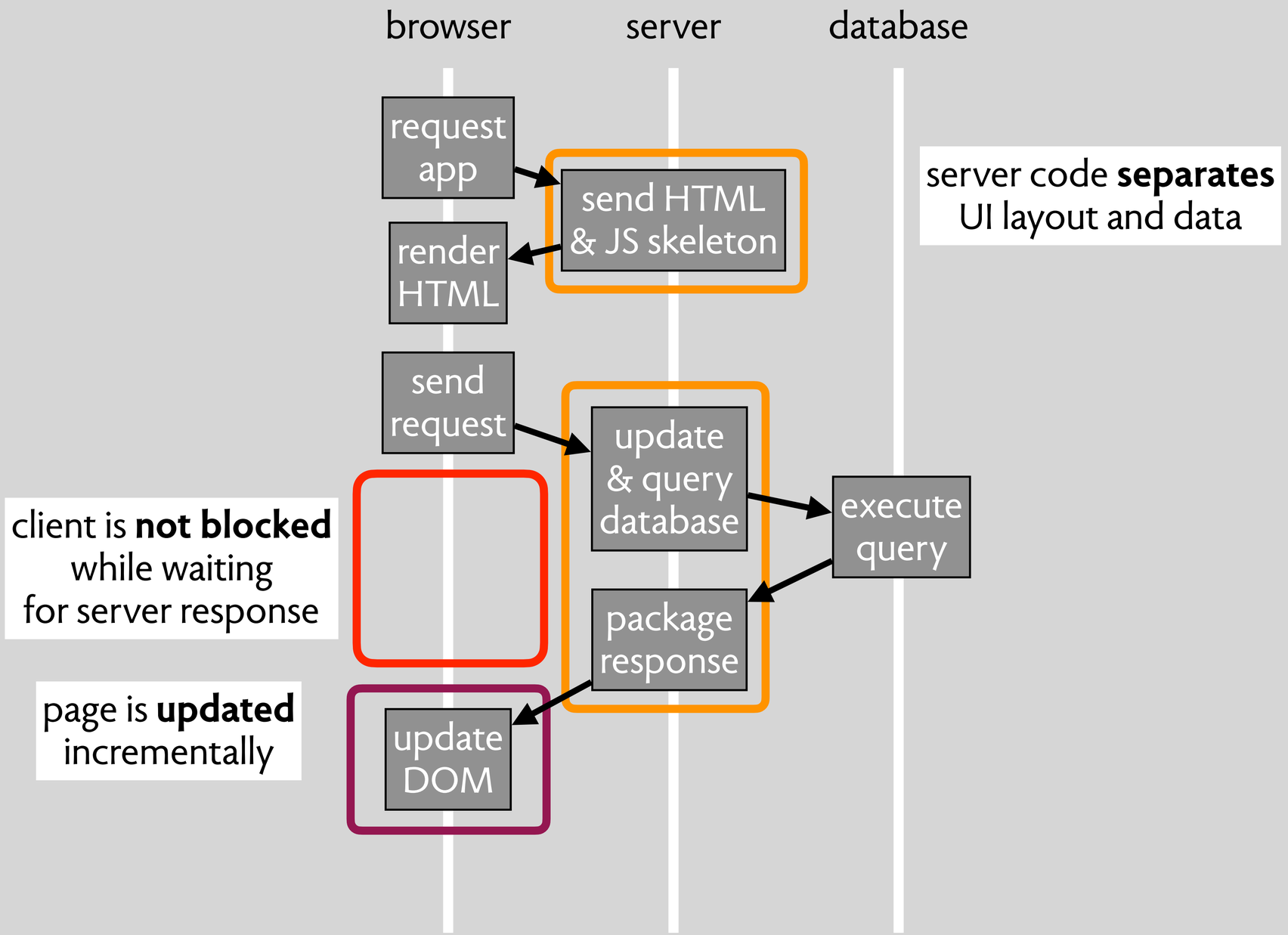
app.py

```jinja
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>{{ user.name }}'s Profile</title>
</head>
<body>
  <h1>Welcome, {{ user.name }}!</h1>
  <p>Age: {{ user.age }}</p>

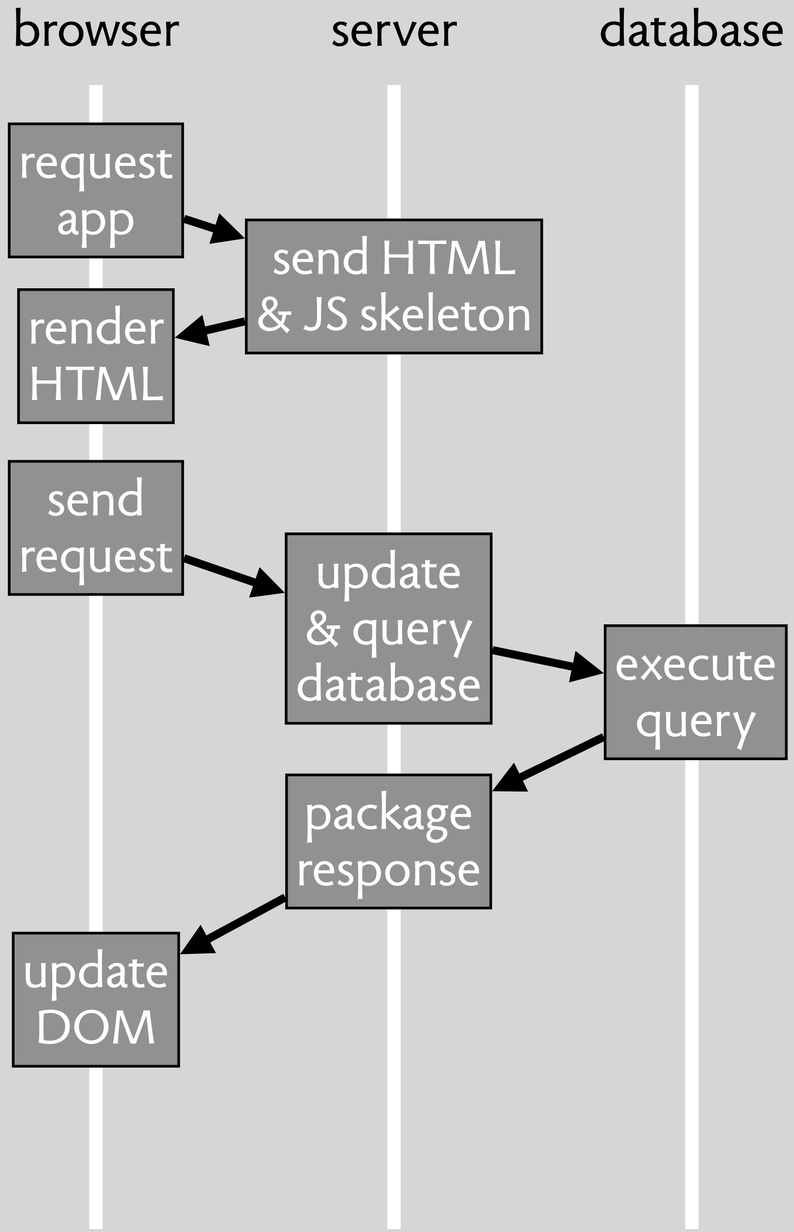
  <h2>Hobbies</h2>
  <ul>
    {% for hobby in user.hobbies %}
      <li>{{ hobby }}</li>
    {% endfor %}
  </ul>
</body>
</html>
```

profile.html

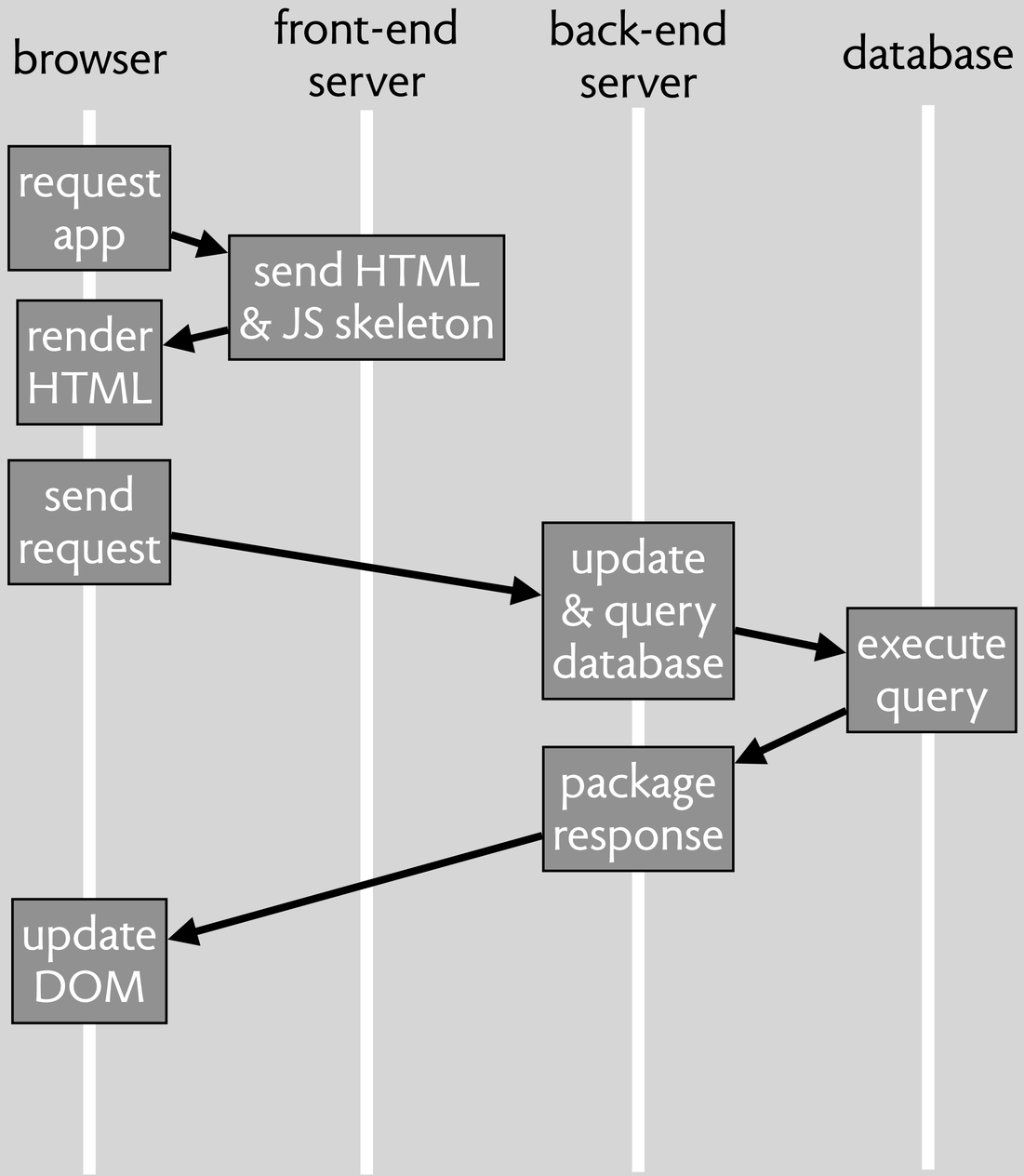
the flow for a “single page” app



the flow for a “single page” app

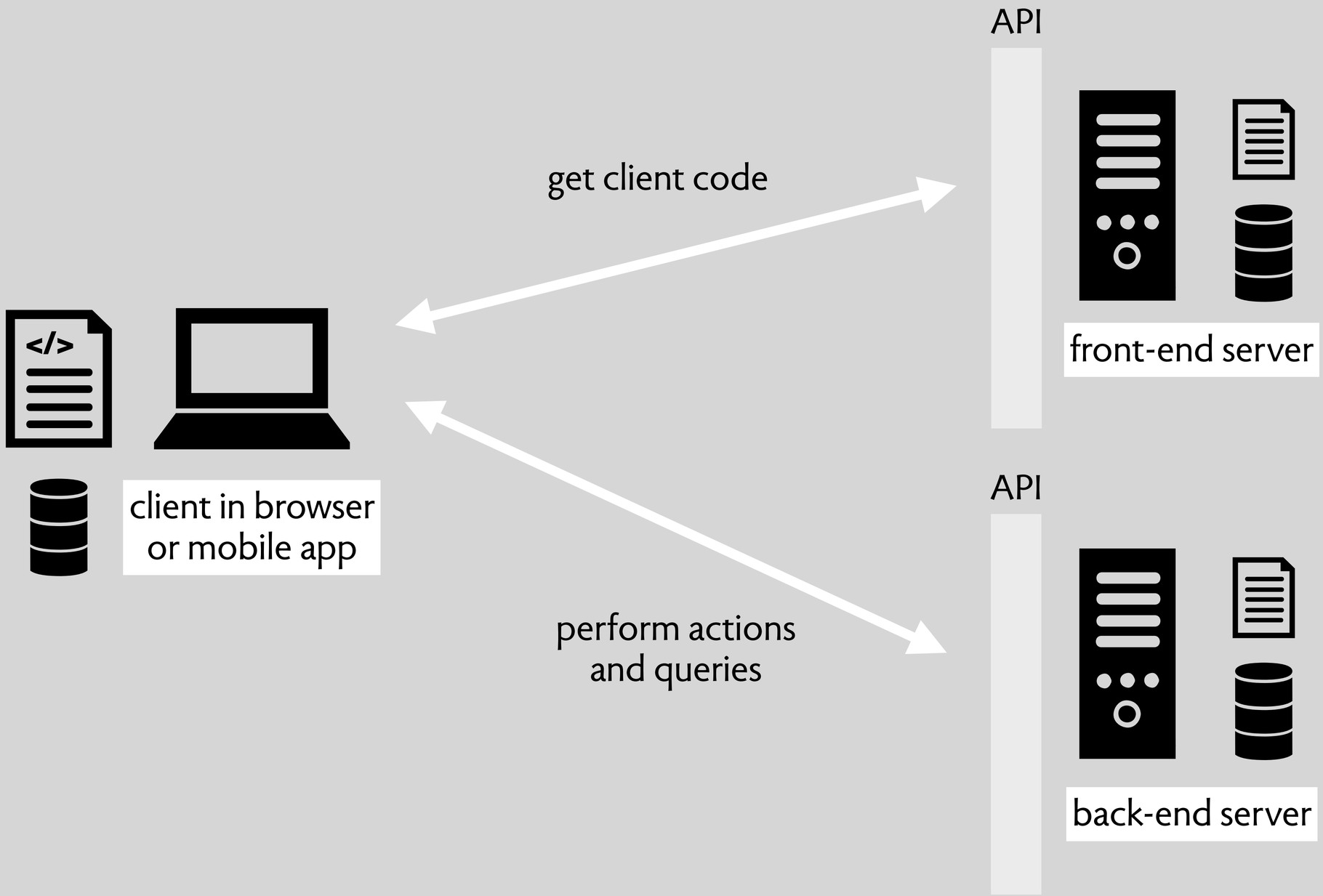


separating the roles of an SPA into two servers



what's an API?

a typical setup



why APIs?
why not just open source the code?

MBTA's API for live transport information

V3 API

The V3 API presents General Transit Feed Specification (GTFS), GTFS Realtime, and MBTA-specific data. It uses the JSON:API format, so you can get started using any of the available libraries.

Skip to any section below:

- Request an API Key
- Make Basic Calls
  - Filter Data
  - Include Relationships
- Use Best Practices
- Learn About API Updates

Request an API Key

We recommend using a V3 API key early in your development process. A valid key is required to make a default of 1,000 requests per minute, control the version, and stream requests.

Request a free key

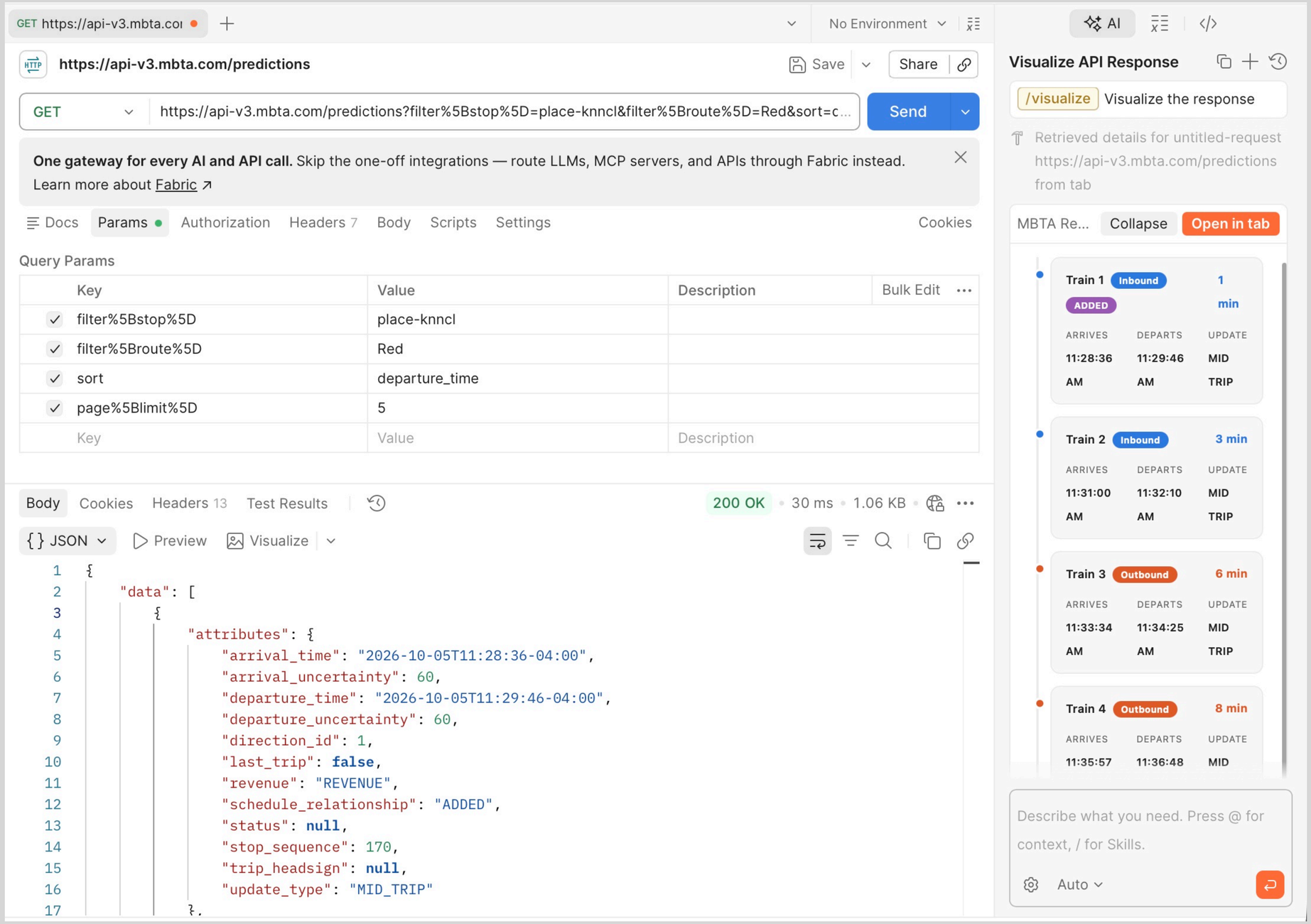
Make Basic Calls

The following V3 API calls are available. To read endpoint documentation for a call, click its link:

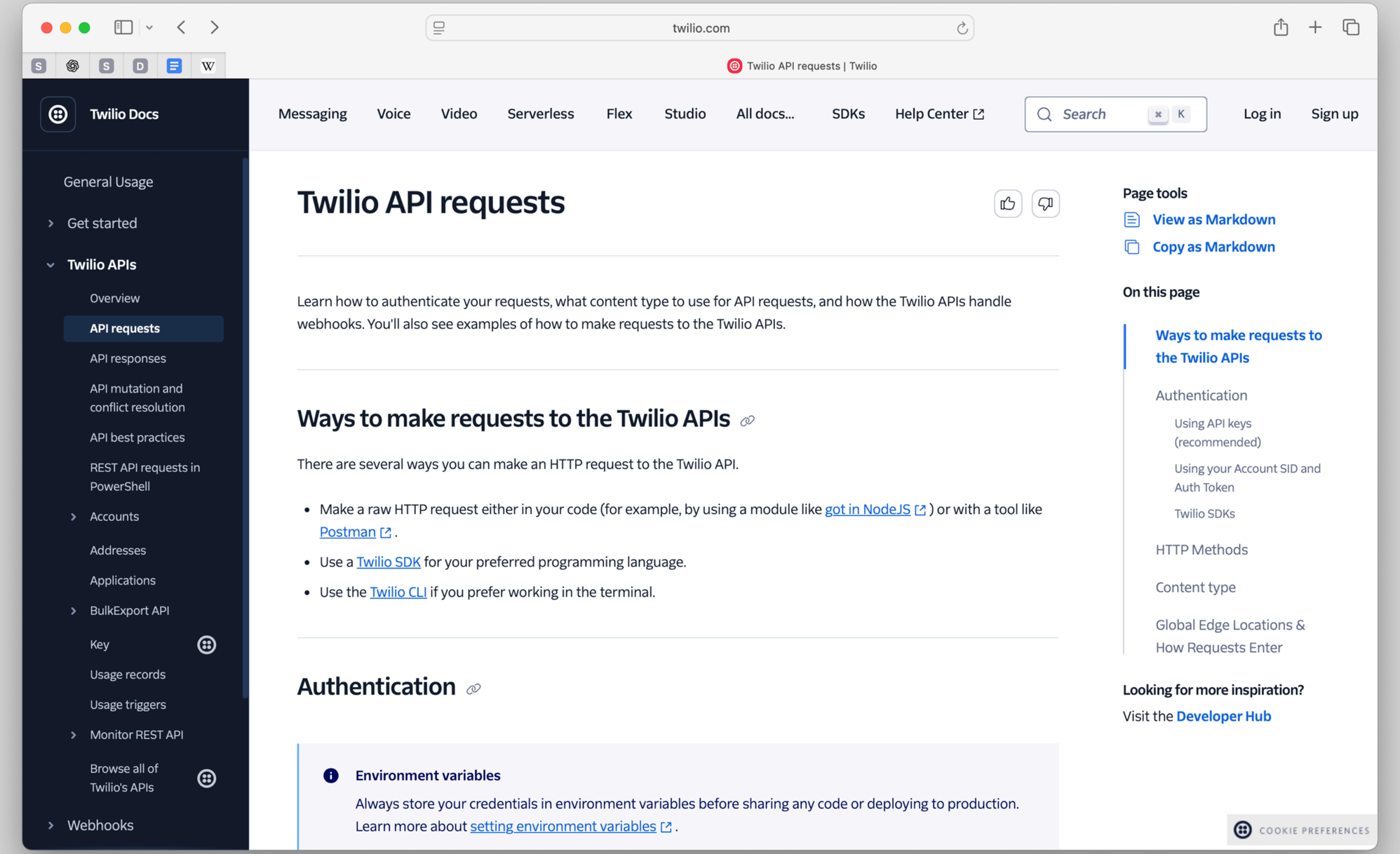
Call	Description
/alerts	Communications to riders about disrupted MBTA operations
/facilities	Amenities at stops, such as elevators, escalators, parking lots, and bike racks
/lines	Lines for all modes (for example, `line-Green`)
/predictions	Predicted arrival/departure times
/routes	Routes for each line (for example, `Green-B`)
/route_patterns	Possible patterns of travel for each route
/schedules	Scheduled arrival/departure times (`stop_time`)
/services	Dates when trips occur
/shapes	Polylines that describe trip shapes, which can be shown on a map
/stops	Places where riders board a vehicle, disembark from a vehicle, or navigate a station
/trips	Movement of vehicles from one route end to the other
/vehicles	Vehicle positions, including direction, latitude, and longitude

Each call returns the object described. Following the JSON:API standard, you can filter and sort the list in various ways, and you can include other objects with related information.

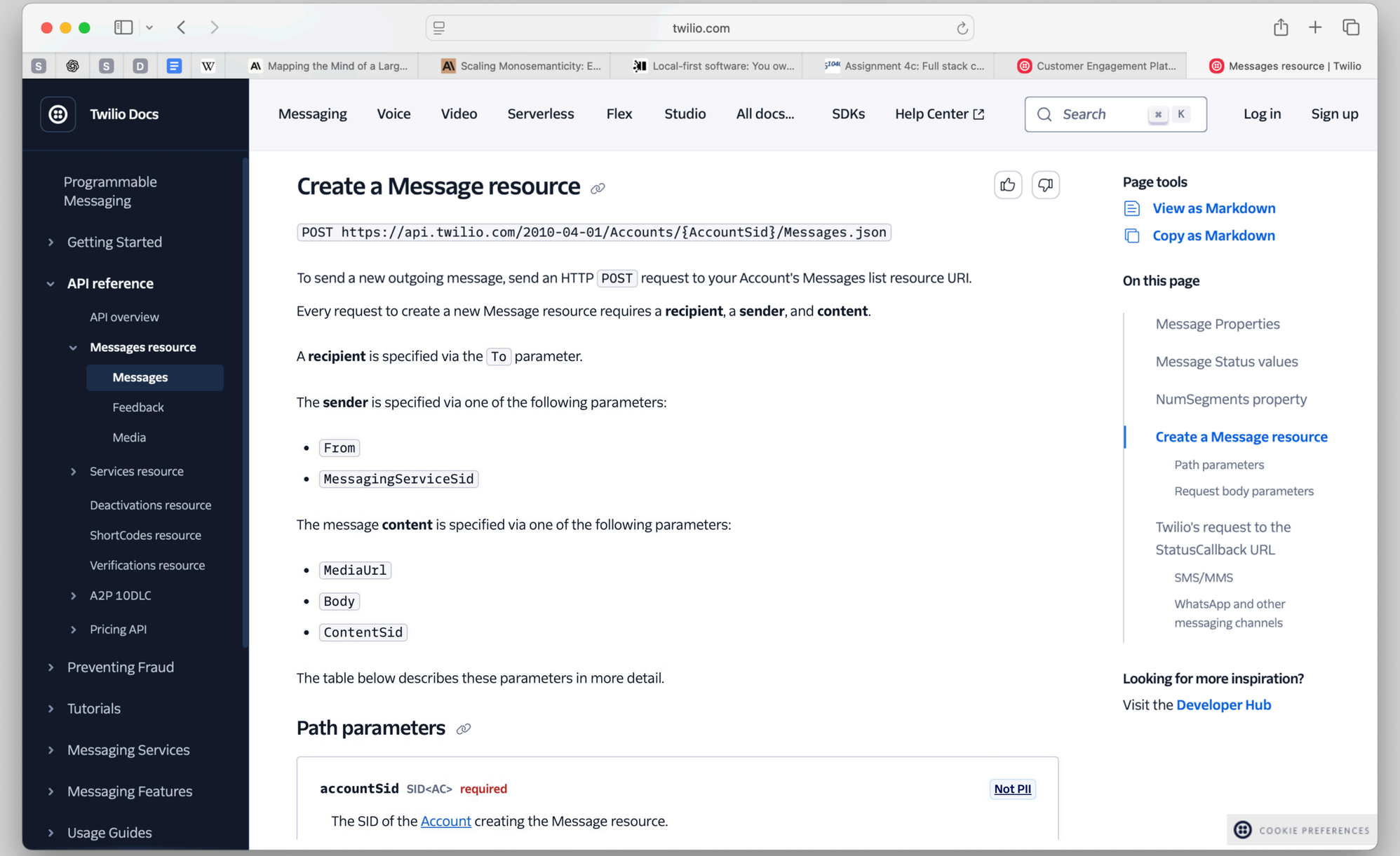
issuing a request in postman



Twilio: a service for sending text messages



creating a message



using the Twilio client-side library

Send an SMS message Node.js

```
// Download the helper library from https://www.twilio.com/docs/node/install
const twilio = require("twilio"); // Or, for ESM: import twilio from "twilio";

// Find your Account SID and Auth Token at twilio.com/console
// and set the environment variables. See http://twil.io/secure
const accountSid = process.env.TWILIO_ACCOUNT_SID;
const authToken = process.env.TWILIO_AUTH_TOKEN;
const client = twilio(accountSid, authToken);

async function createMessage() {
  const message = await client.messages.create({
    body: "Hi there",
    from: "+15557122661",
    to: "+15558675310",
  });

  console.log(message.body);
}

createMessage();
```


```
{
  "account_sid": "ACXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
  "api_version": "2010-04-01",
  "body": "Hi there",
  "date_created": "Thu, 24 Aug 2023 05:01:45 +0000",
  "date_sent": "Thu, 24 Aug 2023 05:01:45 +0000",
  "date_updated": "Thu, 24 Aug 2023 05:01:45 +0000",
  "direction": "outbound-api",
  "error_code": null,
  "error_message": null,
  "from": "+15557122661",
  "num_media": "0",
  "num_segments": "1",
  "price": null,
  "price_unit": null,
  "messaging_service_sid": "MGaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
  "sid": "SMaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
  "status": "queued",
  "subresource_uris": {
    "media": "/2010-04-01/Accounts/ACaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa/Messages/SMaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  },
  "to": "+15558675310",
  "uri": "/2010-04-01/Accounts/ACaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa/Messages/SMaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
}
```

what's a static site?

static web sites

what the user sees

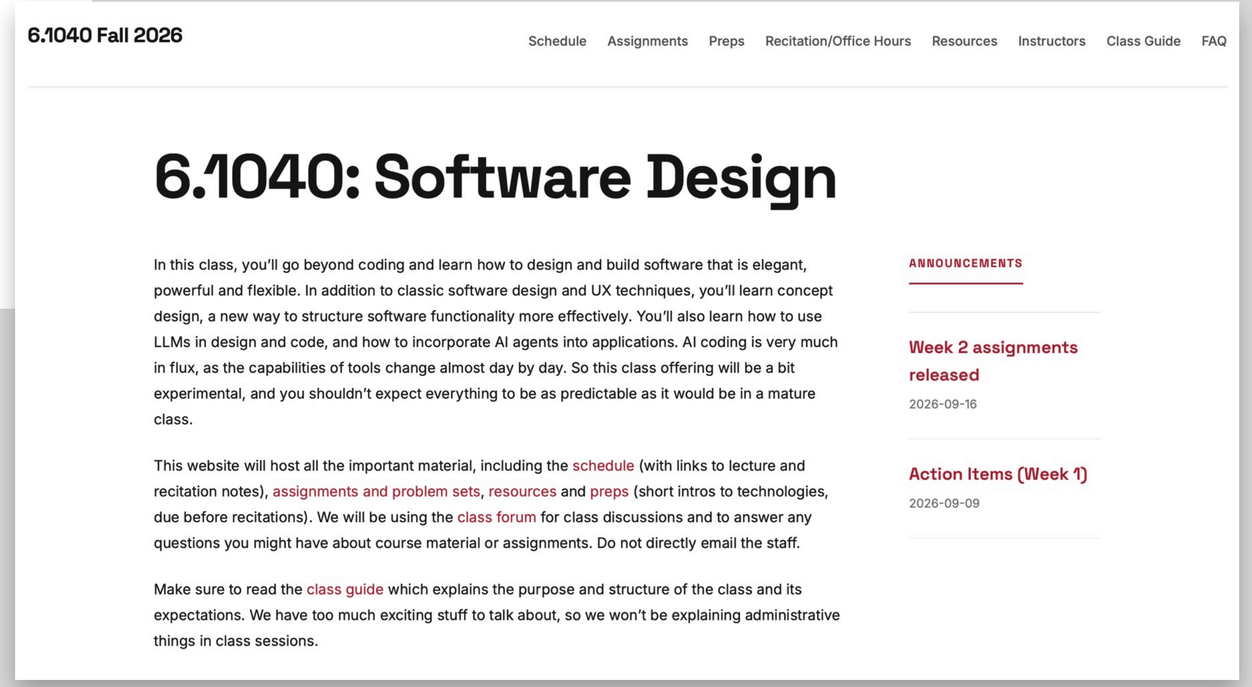
resource returned depends on HTTP request
and not on previous HTTP requests

what the developer sees

site is just a collection of files, not programs or databases
served directly by a web server
can include JavaScript that runs in the browser

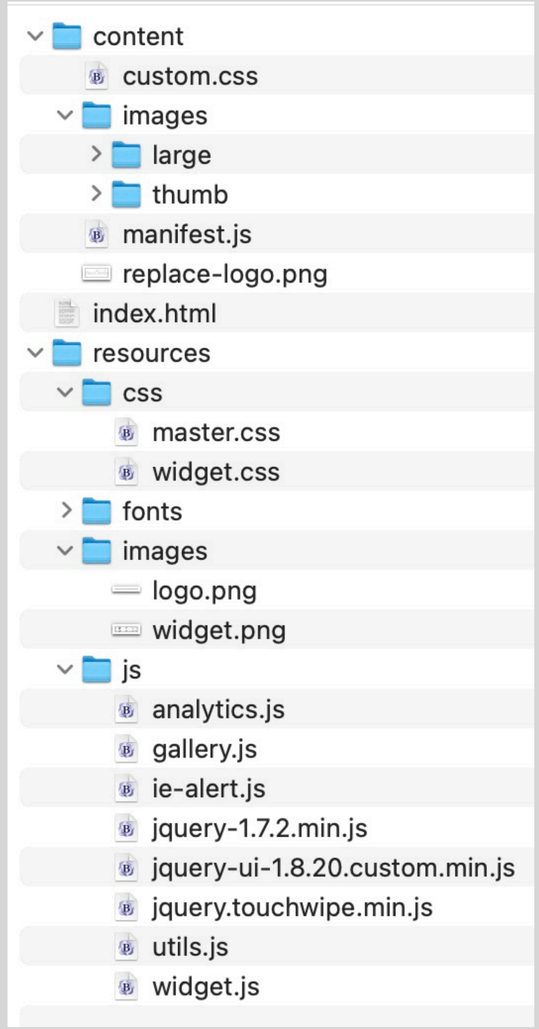
how to create a static site

set up a server or find a hosting service (eg, GitHub Pages)
edit files locally and push to server (eg, with Git or Unison)
or, use a static site builder or CMS



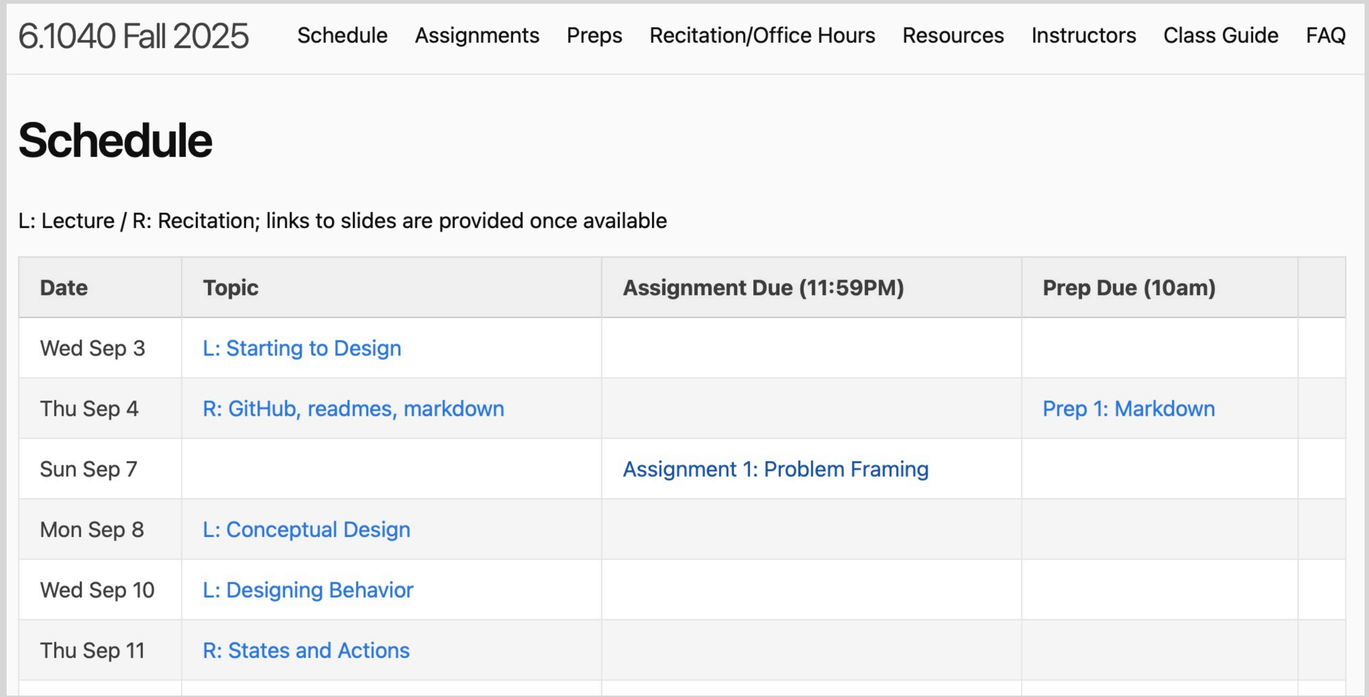
example: my photo web pages





files that produce this page

static website builders





what they offer

editing in markdown, not HTML
consistent styles, packaged in themes
index pages (eg, of blog posts by date)
easy management of nav bars
live view while editing locally

how they work

you edit in markdown
run build command
push to repo

popular examples

Jekyll (Ruby)
Hugo (Go)

disadvantages

sometimes obscure, can be limiting
themes aren't truly interchangeable

content management systems

what they are

static to users, dynamic to admins

updates via HTTP requests (ie, edit site in browser)

examples

WordPress: popular CMS for small sites

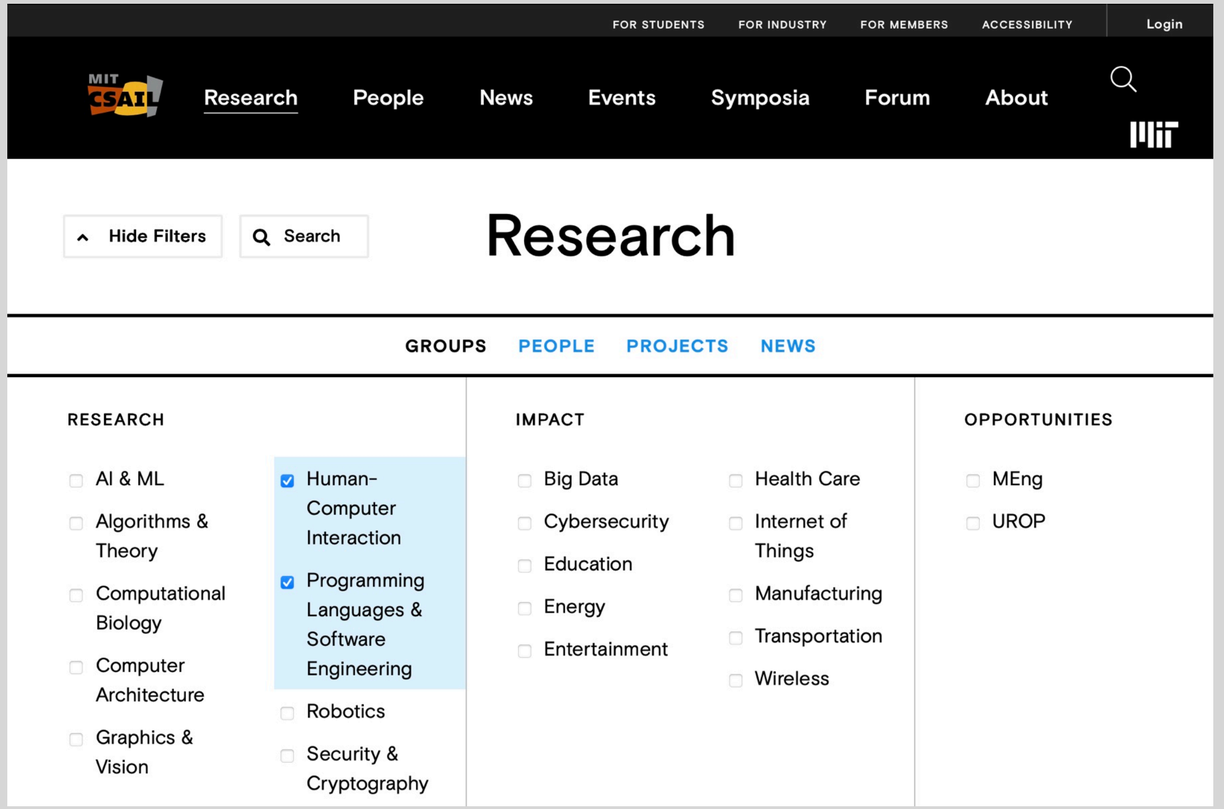
Wix, Google Sites: combined site builder and CMS

SquareSpace: site builder/CMS for small businesses

Drupal: CMS for complex sites, often built by pros



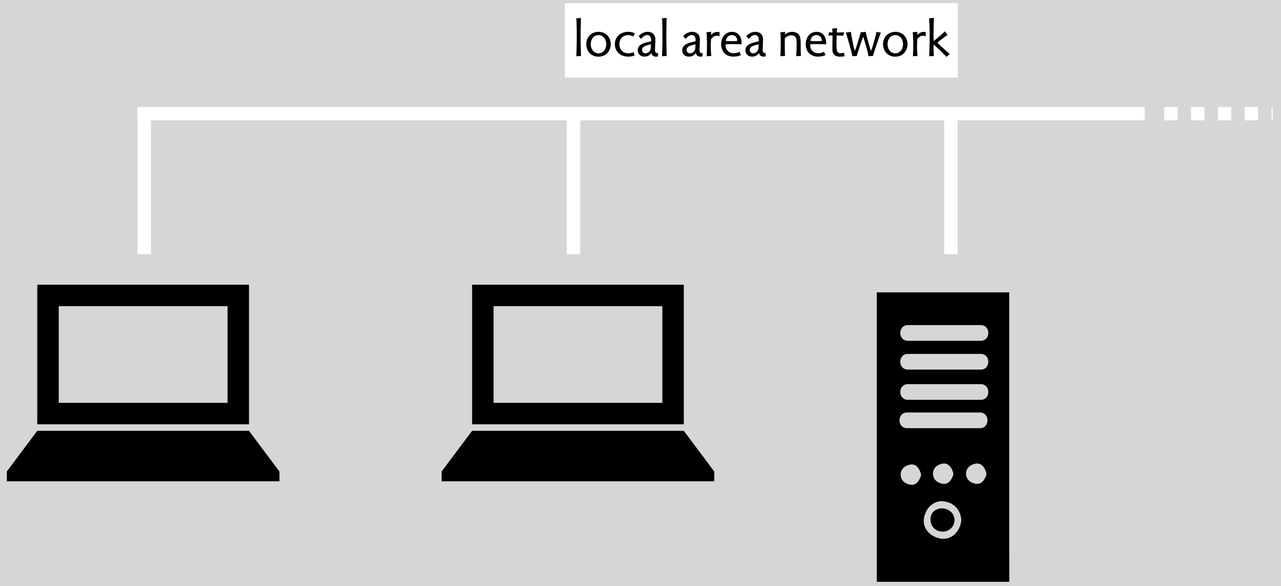
techcrunch.mit.edu (WordPress)



csail.mit.edu (Drupal)

how are clients authenticated?

how to authenticate clients?

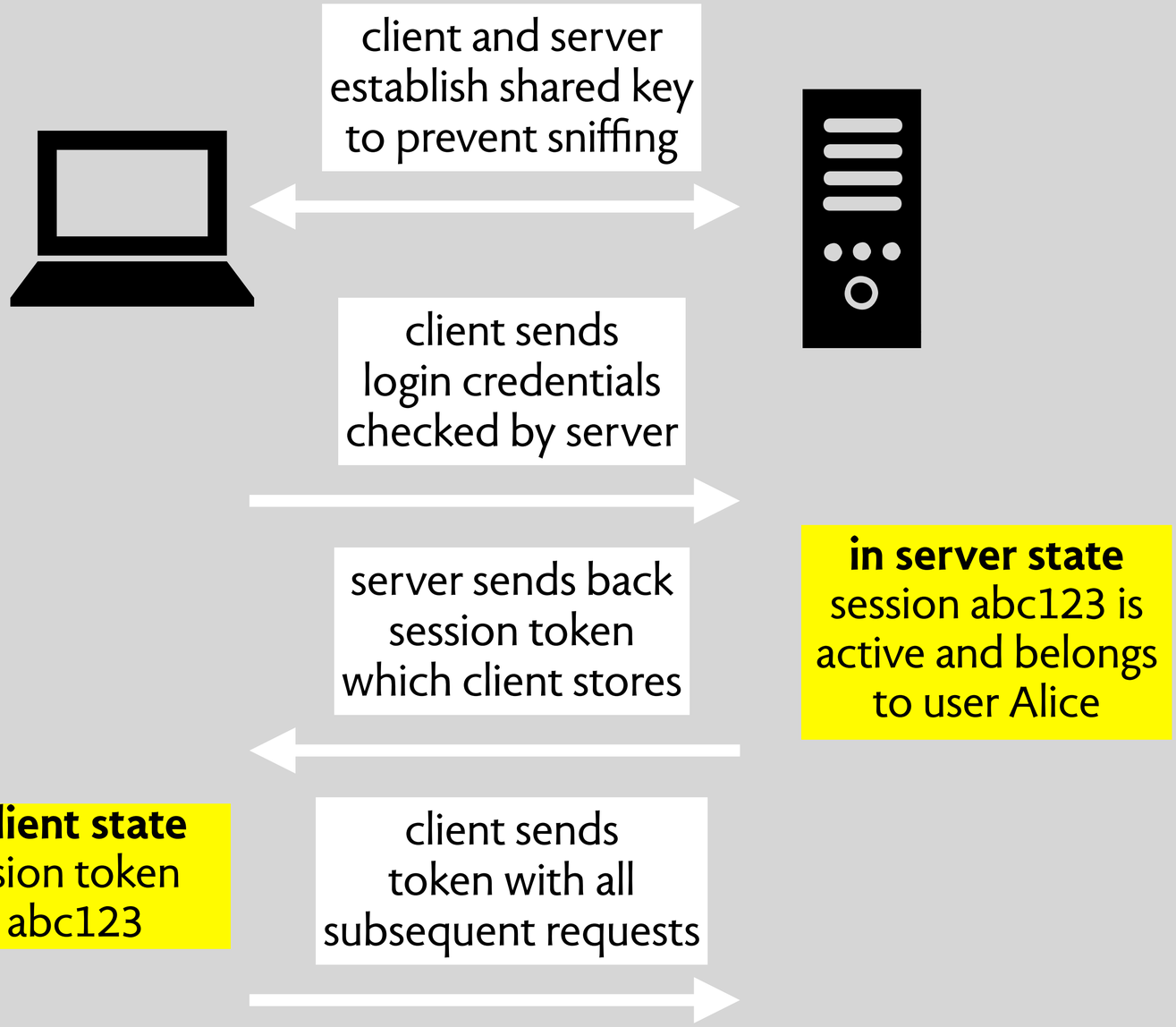


"trust by wire" in early LANs
if you could access the physical network
you were considered authenticated

password auth in early LANs
server opens connection, user logs in & access files
passwords sent in clear, easy to hijack connection too

dial-up modems were a bit safer
client gets next available modem line
modem connection was a private circuit

a better scheme: session tokens



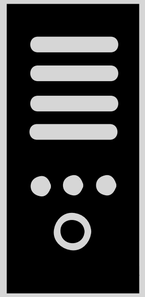
cookies: a way to store session tokens



login (“dj”, “password”)

Set-Cookie: id=abc123;
Domain=nytimes.com

get (nytimes.com/news)
Cookie: id=abc123

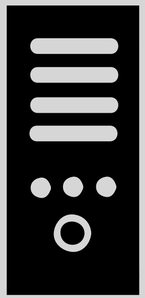


nytimes.com

login (“dj”, “password”)

Set-Cookie: id=def456;
Domain=instagram.com

get (instagram.com/feed)
Cookie: id=def456



instagram.com

other uses of cookies
shopping carts eg

nice for programmer
cookies sent by default

security model
domains stay separate

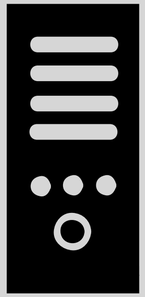
third-party cookies



get (nytimes.com/news)

get (analytics.com/pixel.gif)

Set-Cookie: id=xyz123;
Domain=analytics.com

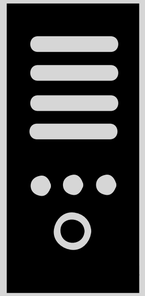


nytimes.com

get (instagram.com/feed)

get (analytics.com/pixel.gif)

Cookie: id=xyz123

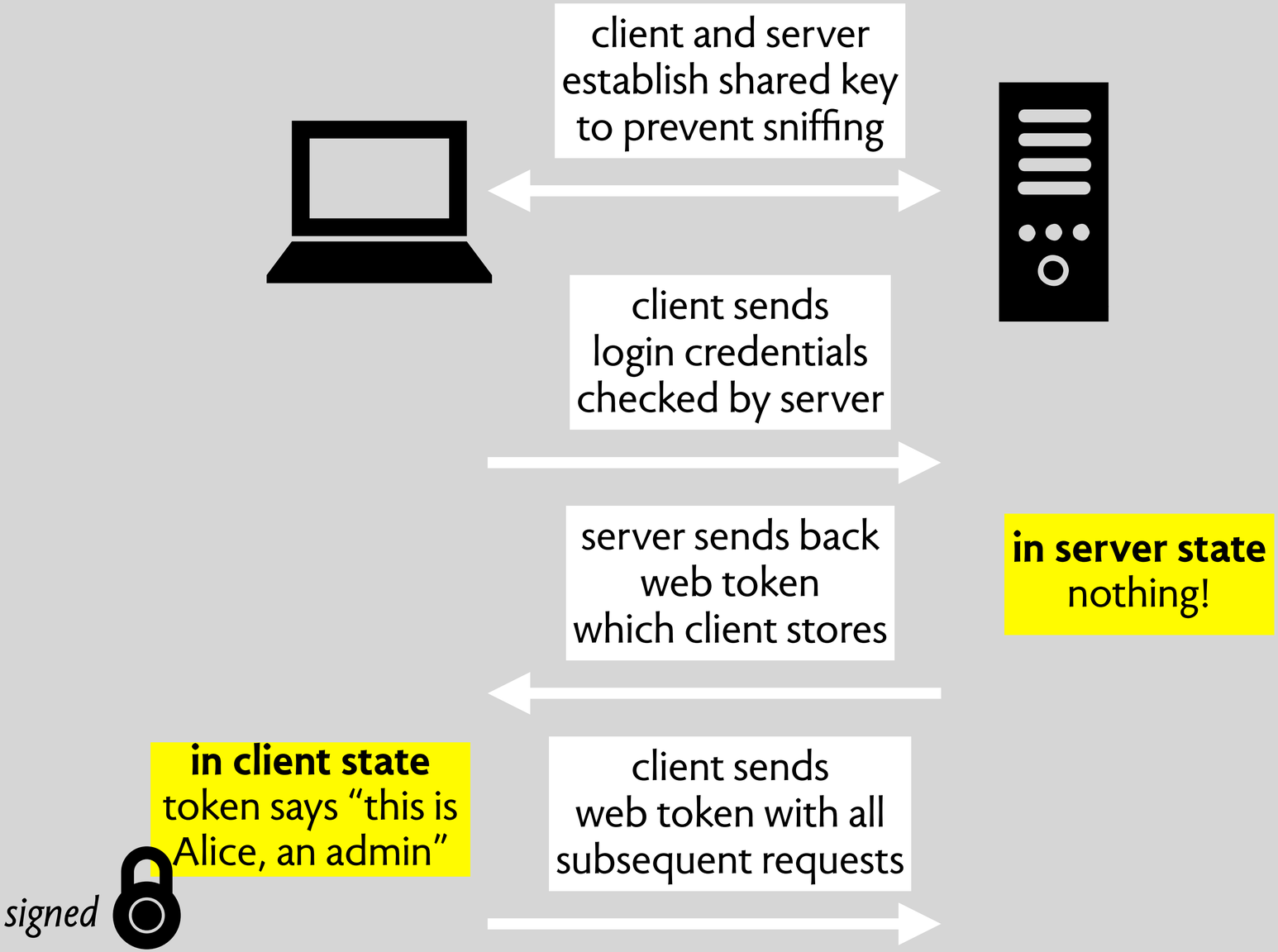


instagram.com

what can this be used for?

cross site tracking
GDPR requires consent

JSON web tokens (JWTs)



refreshing web tokens

a problem: long or short?
with conventional sessions, server can always close a session
but with JWTs, can access until token expires
insecure if token is long lived, but inconvenient if short lived!

solution: two kinds of token
long-lived refresh tokens: rarely used, safe in browser (eg 14 days)
short-lived access tokens: used for all accesses (eg 10 mins)
when access token expires, client uses refresh token to get a new one
when refresh token expires, have to log in again

what goes on the client or server?

security considerations

code and data in the browser
are visible to and modifiable by the user
with developer tools

user can issue any HTTP requests
by modifying JS in the browser document
by commands in the browser JS console
by using curl or Postman

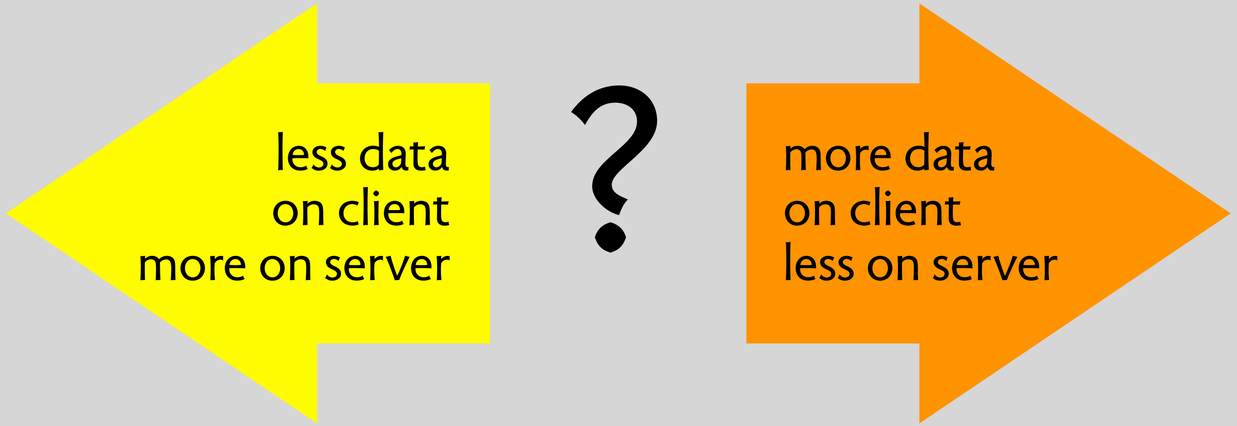
so which of these are good strategies?

to prevent access to another user's data
have client code pass user name with request ✗

to prevent access to sensitive pages
navigate first through login page ✗

to prevent access to another user's data
autoincrement session ids and store in cookie ✗

to prevent access to another user's data
generate random session id and store in cookie ✓


less data
on client
more on server
?
more data
on client
less on server

performance considerations

which decision does it favor?

speed of queries

scaling to more clients

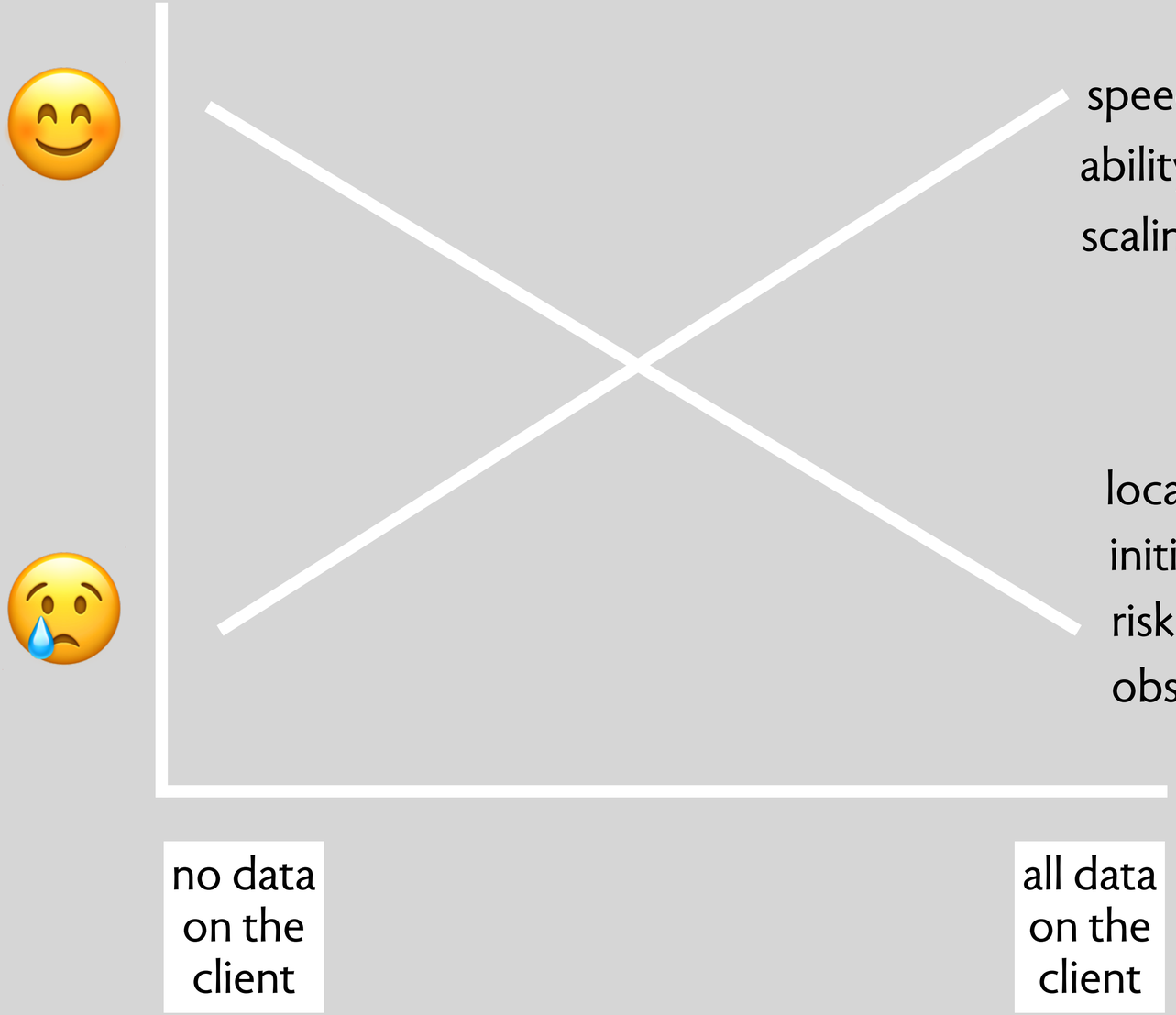
initial startup time

local storage usage

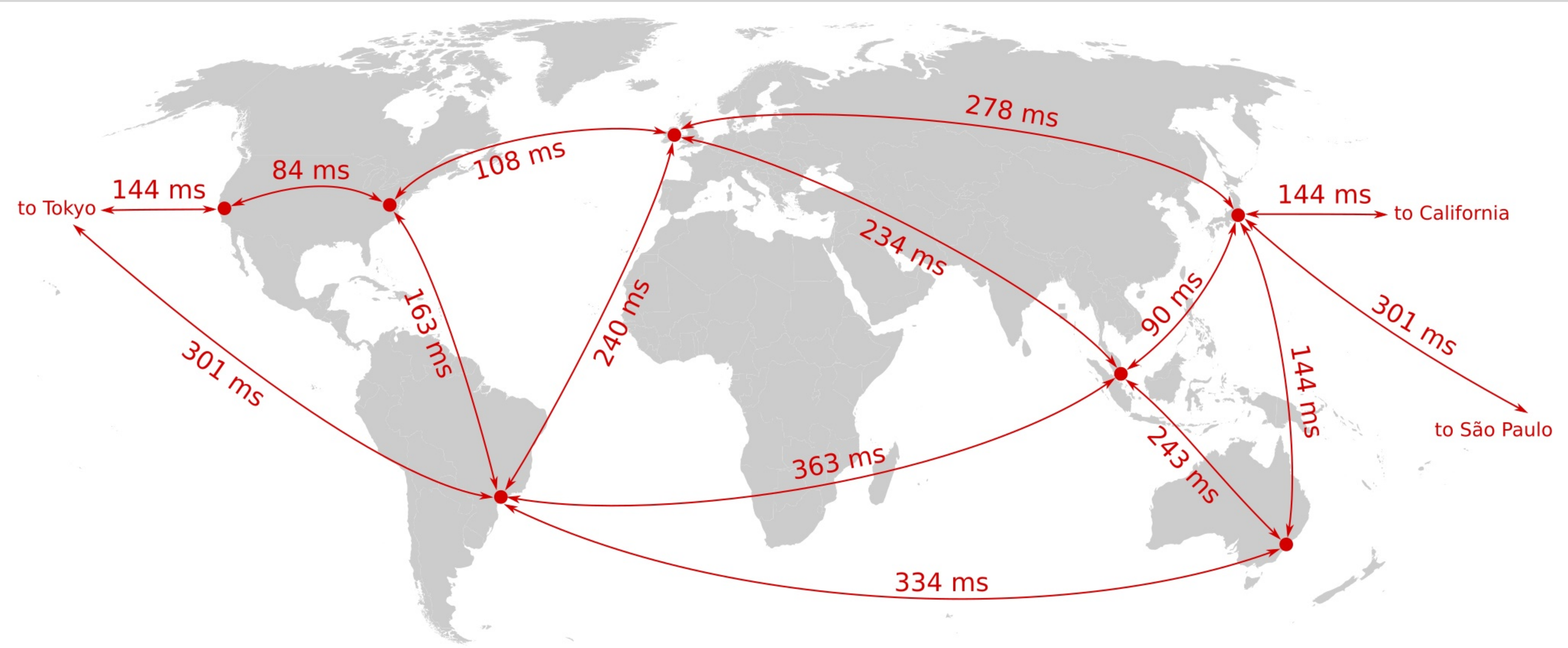
risk of privacy violation

observability by devs

ability to work offline

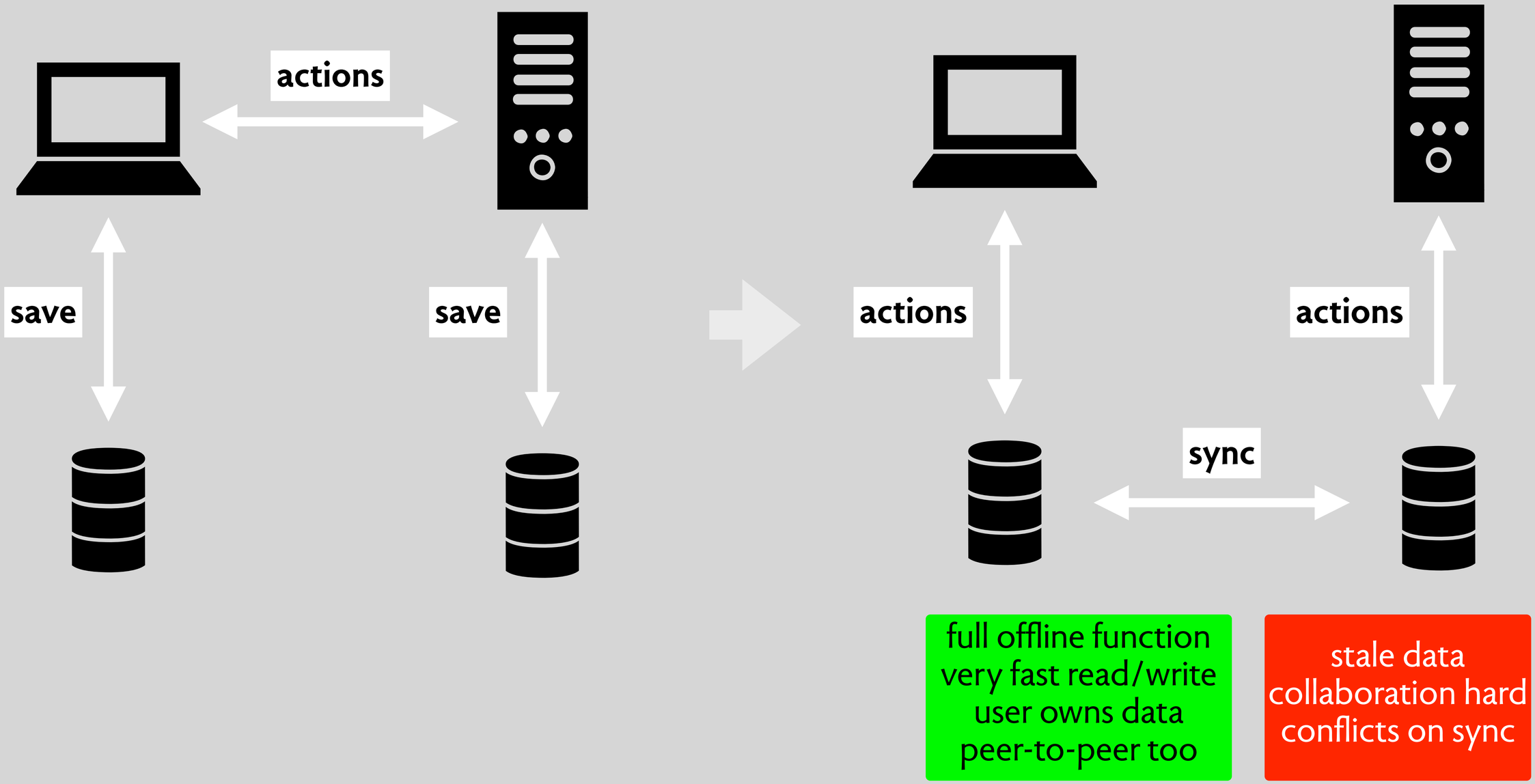


online apps are slow!



server-to-server round trip times between AWS data centers (Ink & Switch)

local first: a proposal for a new kind of app



parting thought

how does AI affect client-server computing?

LLM queries are stateless
each question in a chat sends the entire chat to the LLM
could put entire chat in every HTTP request: ouch!
instead, send just the question and hold chat in server

upshot
reduces traffic between you and the server
but lots of traffic between server-side database and LLM

hence "edge AI"
trying to run LLMs (eg, Apple Siri) on client devices
a tradeoff between performance and quality