

Data Acquisition, Processing and Mining for AI

Week 03 — Web APIs

Silviu Maniu

Getting data off the web

acquire → understand → clean → mine

- Weeks 1, 2: the data was already obtained
- Today: where it actually comes from → a server over HTTP
- Your project's dataset is acquired this way from an API, using a script

A title ending in (*) marks optional material: context or an extension, dropped first if we run short.

Ways to get data

	output	problems
file / dump	everything, once	staleness
API	data according to contract	versioning, quotas, auth
scraping	Web page content	page changes, redesigns

- **API** has a **contract**: URLs, parameters, JSON shape, rate limit
- Scraping has **no contract** → every change to the site changes output
- **Projects are API-based**

HTTP

How machines on the Internet talk

- Your program is the **client**; the machine holding the data is the **server**. The client always speaks first
- **DNS** turns the name `openlibrary.org` into an **IP address**; the address plus a **port** (443) says **which program** on that machine to reach
- **TCP** carries the bytes reliably and in order; **TLS** encrypts them → that is the **s** in `https`
- On top of all that sits **HTTP**: the rules for asking and answering
- What comes back is a **representation** of the resource: **HTML** — the markup a **browser** lays out and renders — or **JSON** — data a **program** parses. Same protocol, different **payload** (the content carried in the body)

RFCs: the “standards” of the Web (*)

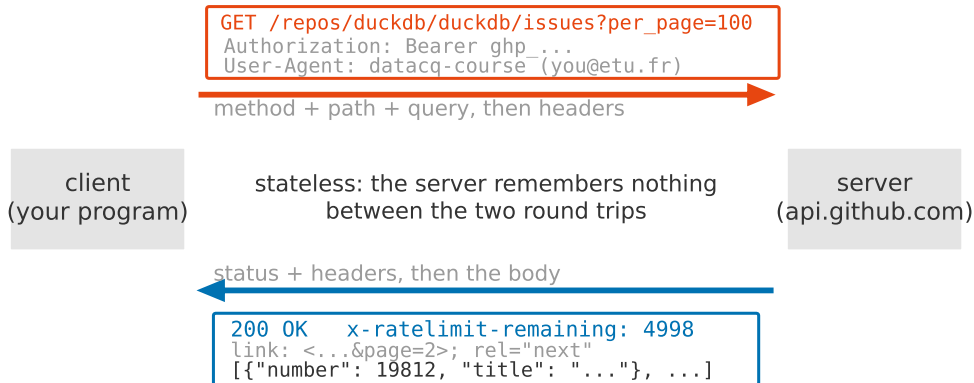
- **RFC** = “Request for Comments”. RFC 1, April 1969, ARPANET: notes on host software, written by a graduate student who picked a **deliberately humble** title
- The name stayed, and became **the standards**. The **RFC Editor** publishes them in **one numbered sequence**, now **past 10 000**
- Never edited, a correction is a new RFC that **obsoletes** the old one
- HTTP: **RFC 2616** (1999) → 7230–7235 (2014) → **RFC 9110** (2022); still evolving: RFC **10008**, June 2026, adds the QUERY method
- Not all of them are standards, and one per year is a joke: RFC **2324** (1 April 1998) specifies HTTP for **coffee pots**: why 418 I’m a teapot exists
- **Free, public, and definitive**. Compare to the SQL standard: ISO/IEC 9075 and has **a cost**

A book, as HTML

```
<h1 class="work-title" itemprop="name">Hands-On Machine  
  Learning with Scikit-Learn, Keras, and TensorFlow</h1>  
<h2 class="edition-byline">  
  by <a href="/authors/OL3897679A/Aurélien_Géron"  
    itemprop="author">Aurélien Géron</a>  
</h2>
```

- **HTML** consists of **nested tags**: an opening <h1>, the content, a closing </h1>. class and id are labels that define page style
- The data: **text between the tags** → reading it means knowing **where in the page** it sits
- The HTML page is **90 kB** of markup. /works/OL20709638W.json returns the same book in **1.8 kB**
- One resource, two representations → **prefer JSON via API if available**, not HTML via scraping

Request and response



- **Stateless:** nothing carries over between two calls → every request re-sends your credentials
- That is why your token is on every single request

URL = query

`https://openlibrary.org/search.json?q=duckdb&limit=10&page=2`

<code>https</code>	scheme	how to talk
<code>openlibrary.org</code>	host	who to talk to
<code>/search.json</code>	path	which resource
<code>?q=duckdb&limit=10&page=2</code>	query string	the parameters

- key=value, joined by &, after a single ?
- **Never build this string by hand**, unencoded characters change the query

Methods

method	does	safe	idempotent
GET	read a resource	yes	yes
POST	create, or “run this”	no	no
PUT	replace a resource	no	yes
PATCH	modify part of it	no	no
DELETE	remove it	no	yes

- **Safe** = changes nothing on the server; **idempotent** = doing it twice equals doing it once
- **Acquisition is GET** → safe to repeat, safe to retry, safe to cache

Status codes

200	OK
201 / 204	created / no content
301 / 302	moved
304	not modified: use your cache
400	do not understand
401	who are you?
403	understood, but forbidden
404	no such resource
422	valid syntax, invalid values
429	too many requests
500	server side error
502 / 503	upstream dead / overloaded

- First digit is what matters: 2 fine, 3 elsewhere, 4 client fault, 5 server fault
- 4xx → retrying changes nothing (except 429). 5xx and 429 → retrying is the right response
- 404 is sometimes a polite 403: GitHub answers 404 for a private repo rather than confirm it exists
- A 200 with an error message in the body can happen

Headers

One real answer — GitHub, /repos/duckdb/duckdb/issues:

```
content-type: application/json; charset=utf-8
x-github-api-version-selected: 2022-11-28
x-ratelimit-limit: 60
x-ratelimit-remaining: 59
x-ratelimit-reset: 1789765656
link: <https://api.github.com/.../issues?...&page=2>; rel="next"
```

- Your budget, the next page and which version answered
- You send: Authorization (your token), User-Agent (who you are, with an email), Accept, If-None-Match ("only if it changed")
- Body = data and the headers are the protocol → have to take them into account

One request, three interfaces: the browser

Paste the URL in the address bar, press Enter. The browser sends:

```
GET /search.json?q=subject%3A%22machine%20learning%22&limit=3 HTTP/1.1
Host: openlibrary.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) ...
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

- That is the **whole request**: a method, a path, a query string, headers. The browser's **developer tools** (F12) show it verbatim
- Result is JSON, rendered differently by browsers: **Firefox** pretty-prints it, **Chrome** shows raw text. The fastest way to look at what an **endpoint** (one path the API answers on) returns
- **Not** a way to acquire: **no loop, no retry, no cache** + browser attaches its own headers and **cookies**

One request, three interfaces: curl

```
curl -s -G "https://openlibrary.org/search.json" \  
  --data-urlencode 'q=subject:"machine learning"' \  
  -d 'limit=3' \  
  -H 'User-Agent: datacq-course (you@etu.fr)' -i
```

- -G sends the data as a **GET query string**; --data-urlencode does the **percent-encoding**; -d adds a plain parameter; -H adds a header
- -i prints the **response headers** before the body; -v prints the request too — **this is how you read status codes and rate-limit headers** without writing a program
- One line, no file, no imports → the fastest way to **check an endpoint before coding against it**

Aside: you are looking at how the internet talks (*)

"curl is used in command lines or scripts to transfer data. curl is also libcurl, used in cars, television sets, routers, printers, audio equipment, mobile phones, tablets, medical devices, settop boxes, computer games, media players and is the Internet transfer engine for countless software applications in over twenty billion installations. curl is used daily by virtually every Internet-using human on the globe." — curl.se

- 1996: Rafael Sagula writes a 300-line download tool; Daniel Stenberg, for currency rates inside an IRC bot, extends it and takes it over. Renamed curl in 1998, still led by him
- So HTTP is the substrate of the Internet, and curl is the client most of it uses
- Libraries such as Python requests are convenience on top of the same protocol

One request, three interfaces: requests

```
import requests
```

```
r = requests.get("https://openlibrary.org/search.json",  
                 params={"q": 'subject:"machine learning",  
                        "limit": 3},  
                 headers={"User-Agent": "datacq-course (you@etu...fr)"},  
                 timeout=10)  
r.raise_for_status()           # turns 4xx/5xx into an exception  
payload = r.json()             # dict parsed from the body
```

- params= does the percent-encoding
- timeout= is not optional, without it a hung server hangs your program There is no default
- The server cannot tell these three interfaces apart.

REST and JSON

REST conventions

- **REST** = representational state transfer; not a standard just some **conventions** for HTTP APIs
- Resources are **nouns in the path**, collections are plural, e.g. `/repos/duckdb/duckdb/issues/19812`
- The **method** says what to do with them
- Filtering, sorting and paging in the **query string**
- Representations are negotiated Accept: `application/json`
- APIs have **versions**, and the version influences the acquisition
- **Almost nothing on the web is strictly REST**

An example call and response

```
GET https://openlibrary.org/search.json
    ?q=subject:"machine learning"&limit=3
    &fields=key,title,author_name,subject,first_publish_year
{"numFound": 1793, "start": 0, "numFoundExact": true,
 "docs": [
   {"key": "/works/OL20709638W",
    "title": "Hands-On Machine Learning with Scikit-Learn...",
    "author_name": ["Aurelien Geron"],
    "subject": ["machine learning", "Mathematics"],
    "first_publish_year": 2019}]}
```

- numFound is the size of the answer, docs is the slice asked for → 1 793 results, 3 returned
- fields= keeps the response small → ask only for what you need

From JSON trees to tables

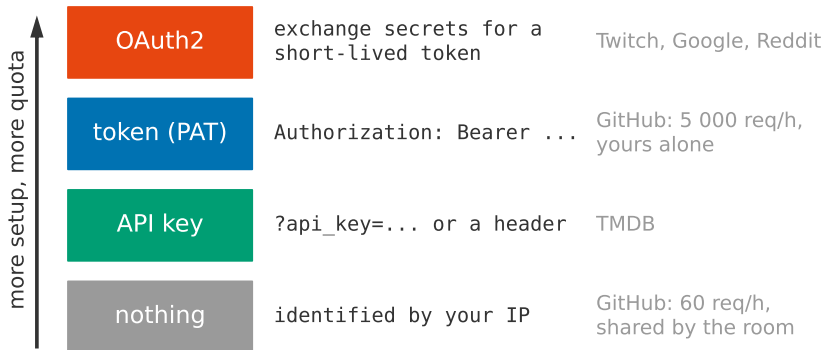
```
import pandas as pd
df = pd.json_normalize(payload["docs"])    # flattens nested keys
df = df.explode("subject")                # one row per (work, subject)
```

key	title	author_name	subject
OL20709638W	Hands-On ML	[Geron]	machine learning
OL20709638W	Hands-On ML	[Geron]	Mathematics
OL17801809W	Deep Learning	[Goodfellow, ...]	Machine learning

- `json_normalize` flattens **nested objects**; `explode` turns a **list-valued cell** into rows
- “**One row per what?**” is a modelling decision: per work, or per (work, subject)?
- Answer it before doing acquisition

Authentication

Levels of authentication



- Every rung answers one question: **who is the requestor and what is their quota?**
- Unauthenticated → the answer is **“this IP’s”**, and the IP is shared

Why not unauthenticated?

- GitHub, unauthenticated: 60 requests per hour, counted per IP address
- A shared unauthenticated approach is one IP!
- With a personal token: 5 000 requests per hour, only for the person owning the token
- The token is not about permission but about being counted individually

Creating your token: GitHub

1. <https://github.com/settings/personal-access-tokens> → **Generate new token**
2. Name it for the job: datacq-2026
3. **Expiration: 90 days**
4. Repository access: **Public repositories (read-only)**
5. Permissions: **none. Ask for the least that works**
6. Generate, then **copy it**: GitHub shows it **once**

Where to keep tokens

```
import os, getpass
token = os.environ.get("GITHUB_TOKEN") or getpass.getpass("PAT: ")

r = requests.get("https://api.github.com/rate_limit",
                 headers={"Authorization": f"Bearer {token}",
                          "Accept": "application/vnd.github+json",
                          "X-GitHub-API-Version": "2022-11-28"},
                 timeout=10)
r.headers["x-ratelimit-limit"]      # 60 without, 5000 with
```

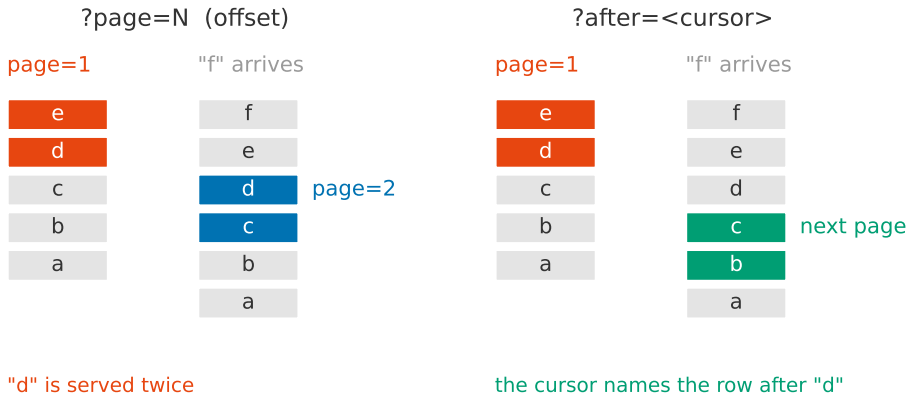
- **Not in your code.** An environment variable, or getpass
- Bearer: **whoever presents the token is authorized**, nothing else is checked!
- **Never in the URL:** URLs are logged by every machine on the way
- A notebook saves its outputs to disk → **never print(token)**

```
r = requests.post("https://id.twitch.tv/oauth2/token",
                  data={"client_id": CLIENT_ID,
                        "client_secret": CLIENT_SECRET,
                        "grant_type": "client_credentials"},
                  timeout=10)
tok = r.json()    # {"access_token": ..., "expires_in": 5011271, ...}
```

- **Client credentials**: the **app** authenticates as itself. **No user is involved**
- The long-lived **secret** goes to one endpoint only; every later request is the **short-lived token** → Authorization: Bearer <access_token>
- expires_in is seconds → **re-fetch before it dies**, do not assume it lasts forever

Pagination

No server gives you everything at once



- **Offset** (page, offset, start): simple, but can be wrong on a moving collection
- **Cursor** (after, since, next_token): stable, you cannot jump to page 7

Offset paging

```
page, rows = 1, []  
while True:  
    p = get_json(URL, {"q": QUERY, "limit": 100, "page": page})  
    rows += p["docs"]  
    if len(rows) >= p["numFound"] or not p["docs"]:  
        break  
    page += 1
```

- Stop on **two** conditions: the count is reached, **or** the server returned an empty page
- Sort by something **stable** (a creation date, an id) if the API lets you

Link-header paging

```
url = "https://api.github.com/repos/duckdb/duckdb/issues"
params, rows = {"per_page": 100}, []
while url:
    r = requests.get(url, headers=H, params=params, timeout=10)
    rows += r.json()
    url = r.links.get("next", {}).get("url")    # None at the end
    params = None                               # the next URL already carries per_page
```

- The server sends the next URL in a Link header (**RFC 8288**); `r.links` is requests parsing it for you
- **Not a third kind of paging**: it is the server handing you the next **query**.
- For example, a plain `?page=2`; GitHub uses `page`, `before/after` or `since` depending on the endpoint
- **Never rebuild that URL yourself**
- The loop ends when the header stops coming. No counting needed

Cursor paging

```
p = {"filter": "issn:2624-8212", "rows": 20, "cursor": "*",  
     "mailto": EMAIL}           # polite pool: 3 req/s, no key  
rows = []  
while True:  
    m = requests.get(CROSSREF, params=p, timeout=10).json()["message"]  
    if not m["items"]:  
        break  
    rows += m["items"]  
    p["cursor"] = m["next-cursor"]
```

- It never stops coming. Past the last record the API keeps returning a cursor and an empty items → a loop waiting for the cursor to vanish never terminates
- Opposite of the Link header, where the absence of next is the signal

Issues with pagination

```
GET /search/repositories?q=duckdb&per_page=100&page=11
```

```
HTTP/2 422
```

```
{"message": "Only the first 1000 search results are available"}
```

- **Silent caps.** GitHub search: pages 1–10 came back like any other, and the cap appears at **page 11**
- **Deep offsets get slow or refused**; page=5000 is not a supported query on most APIs
- The total may move while you page
- Log **what you asked for** (query, dates, page size, when)

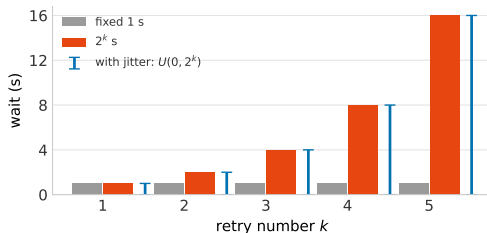
Rate limits and failure

Budgets and rate limits

```
r.headers["x-ratelimit-limit"]      # 5000
r.headers["x-ratelimit-remaining"]  # 4998
r.headers["x-ratelimit-reset"]      # 1785429938 (epoch seconds)
```

- **Primary limit:** requests per hour. **Secondary limit:** bursts, concurrency, expensive endpoints
- Over the limit → 429 (GitHub also uses 403; retry **only** when `x-ratelimit-remaining` is 0; otherwise it is a real refusal), with **Retry-After** in seconds
- **Honour Retry-After.** RFC 9110 allows **seconds or an HTTP date**
- Open Library: **1 request/s, 3/s** if your User-Agent names you and your email

Backoff and jitter



- Fixed retries **keep hammering** a server that is overloaded
- If the failure is **shared** (a rate-limit window resetting, a restart, a deploy) $\rightarrow$ all N clients return at $t_0 + t$ having **the same collision**
- **Jitter**: each client draws $D_i \sim \mathcal{U}(0, T)$ **independently**, with the window $T = \min(\text{cap}, b 2^k)$ **doubling** per attempt k from a base b , up to a **ceiling** cap
- Arrivals in a window of width δ become $\text{Bin}(N, \delta/T)$, expected count $N \rightarrow N\delta/T$, a **Poisson stream of rate N/T** for large N

Jitter fundamentals (*)

- **ALOHA** (Abramson 1970) resends a collided packet after a **random** delay. **Ethernet** (Metcalfe & Boggs 1976) adds **binary exponential backoff**: a random multiple of the slot time, with the range **doubled** after each collision
- **In a standard**: CoAP (RFC 7252) draws the first timeout from $\mathcal{U}(2\text{ s}, 3\text{ s})$ and tells implementers to keep the factor away from 1.0 “to provide some protection from **synchronization effects**”
- **Measured** by Brooker (AWS, 2015) on 100 contending clients: full jitter **more than halves** the number of calls **and** finishes sooner.

A network-aware loop

```
def fetch(url, p, attempts=5):
    for k in range(attempts):
        wait = random.uniform(0, 2**k)
        try:
            r = requests.get(url, params=p, headers=H, timeout=10)
            if r.status_code == 200:
                return r.json()
            if r.status_code not in (429, 500, 502, 503, 504):
                r.raise_for_status() # 400/404
            wait = int(r.headers.get("Retry-After", 0)) or wait
        except requests.RequestException: # timeout, reset, DNS
            pass
        time.sleep(wait)
    raise RuntimeError(f"gave up on {url}")
```

- A timeout or a dropped connection **never reaches status_code**
- **Give up loudly**: every retry loop needs a ceiling

Caching responses

```
key = hashlib.sha256(f"{url}{sorted(params.items())}"
                    .encode()).hexdigest()[:16]
path = CACHE / f"{key}.json"
if path.exists():
    return json.loads(path.read_text())
data = fetch_with_retry(url, params)
path.write_text(json.dumps(data))
```

- An acquisition run **will** die (laptop sleeps, wifi drops, quota ends)
- Caching allows **resuming**
- Re-running the script costs **zero requests**
- The cache key is **URL + parameters**

Rules for good clients

- **Identify yourself.** The default is `python-requests/2.34.2`; send the project name and a real email
- **One request at a time.** Parallelism may be blocked by secondary limits
- **Ask for less:** `fields=`, `per_page=100`, `filters`
- If a **bulk dump** exists and fits your application, download the dump

Hidden APIs (*)

Hidden APIs (*)

- Opening a web page fetches the **HTML**, and the HTML then tells the browser to fetch **dozens more things**: images, fonts, stylesheets, code
- That code then makes **its own HTTP requests** for the actual content
- Those calls are named **XHR** or **fetch** (the two browser mechanisms that make them) and what they usually ask for **JSON**
- So the site has **a JSON API**, but not documented anywhere → a **hidden API**

Finding hidden APIs (*)

1. Open the browser's **developer tools**, every browser has them
2. Go to the **Network** panel, which lists **every request the page makes**
3. Filter on **Fetch/XHR** to see **the data calls**
4. **Reload the page**, then click one row → URL, query string, headers, and the result JSON
5. Right-click → **Copy as cURL**: the **exact** call, headers included, as a `curl` command line

Using hidden APIs (*)

- Structured data with no HTML parsing, better for acquisition
- Undocumented means no contract, it can change anytime
- Often session-bound, it only answers if you send the cookies the browser got when the page loaded
- Use it to find out what exists, but build on documented endpoints

Lab and next week

Today's lab

- M0 was due last night
- Three exercises are mandatory, to allow you to practice for M1: client **retries**, a **pagination loop**, and a **stored dataset with a log**
- **Offline**: set OFFLINE=1 and the lab replays captured responses
- Optional, at home: the **GitHub** calls with the token you just created, and the **Twitch** client-credentials exchange

Before next week

- **M1 is due before the week-5 lab**: raw dataset acquired from your source and stored as Parquet
- Write the acquisition script this week and **run it at home**
- **Cache from the first run**
- The first real pull may invalidate your scope **you may re-register the scope once**
- Next lecture: **scraping**, law and ethics, and Parquet details

References

- The RFC Editor — every RFC since 1969, free: <https://www.rfc-editor.org>
- RFC 9110, *HTTP Semantics* (methods, status codes, headers).
<https://www.rfc-editor.org/rfc/rfc9110>
- RFC 8259, *The JavaScript Object Notation (JSON) Data Interchange Format*.
<https://www.rfc-editor.org/rfc/rfc8259>
- RFC 6749, *The OAuth 2.0 Authorization Framework*; RFC 6750, *Bearer Token Usage*.
<https://www.rfc-editor.org/rfc/rfc6749>
- RFC 8288, *Web Linking* (the Link header). <https://www.rfc-editor.org/rfc/rfc8288>
- Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, ch. 5 (REST), 2000. https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- GitHub REST API: <https://docs.github.com/en/rest> — rate limits: <https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api>
- Open Library API: <https://openlibrary.org/developers/api>
- Crossref REST API (the cursor example):
<https://www.crossref.org/documentation/retrieve-metadata/rest-api/>
- requests: <https://requests.readthedocs.io> — curl and its history:
<https://curl.se/docs/history.html>

References: backoff and jitter

- Abramson, *The ALOHA System – Another Alternative for Computer Communications*, AFIPS FJCC, 1970 – random retransmission delay. <https://doi.org/10.1145/1478462.1478502>
- Metcalfe & Boggs, *Ethernet: Distributed Packet Switching for Local Computer Networks*, CACM 19(7):395–404, 1976 – binary exponential backoff.
<https://doi.org/10.1145/360248.360253>
- Aldous, *Ultimate Instability of Exponential Back-off Protocol...*, IEEE Trans. Inf. Theory 33(2):219–223, 1987. <https://doi.org/10.1109/TIT.1987.1057295>
- Goodman, Greenberg, Madras & March, *Stability of Binary Exponential Backoff*, JACM 35(3):579–602, 1988. <https://doi.org/10.1145/44483.44488>
- Brooker, *Exponential Backoff And Jitter*, AWS Architecture Blog, 2015 – full / equal / decorrelated jitter, simulated.
<https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/>
- RFC 7252 Sec. 4.2 and Sec. 4.8 (CoAP) – randomised initial timeout, against “synchronization effects”. <https://www.rfc-editor.org/rfc/rfc7252>