APIs and Data Collection at Scale

Jared Edgerton

Why APIs Matter for Social Data

Many large-scale social datasets are accessed through:

- Application Programming Interfaces (APIs)
- Structured endpoints
- Machine-readable responses

APIs are often the *intended* interface for data access.

APIs Are Contracts

An API specifies:

- What data can be requested
- How requests must be structured
- What constraints apply to usage

APIs encode institutional and technical choices.

APIs vs Web Scraping

Key differences:

- APIs expose structure explicitly
- APIs enforce access rules
- APIs log and monitor usage

Scraping recovers structure; APIs declare it.

Common API Response Formats

Most social-science APIs return:

- JSON objects
- Nested key–value structures
- Lists of records

Understanding structure matters more than syntax.

JSON as a Data Structure

JSON represents:

- Objects (key–value pairs)
- Arrays (lists)
- Nested hierarchies

APIs return *trees*, not tables.

Authentication Patterns

APIs often require:

- API keys
- Tokens
- OAuth flows

Authentication governs:

- Access
- Rate limits
- Attribution

API Requests (Python)

```python
import requests

url = 'https://api.example.com/v1/items'
params = {'limit': 100}
headers = {'Authorization': 'Bearer YOUR_TOKEN'}

resp = requests.get(url, params=params, headers=headers)
data = resp.json()
```

API Requests (R)

```r
library(httr)

url <- 'https://api.example.com/v1/items'
resp <- GET(
  url,
  query = list(limit = 100),
  add_headers(Authorization = 'Bearer YOUR_TOKEN')
)

data <- content(resp, as = 'parsed')
```

Pagination Is the Norm

Large APIs rarely return everything at once.

Common patterns:

- Page numbers
- Offsets
- Cursor-based pagination
- Time-based windows

Pagination Loop (Python)

```python
all_items = []
page = 1

while True:
    resp = requests.get(
        url,
        params={'page': page},
        headers=headers
    )
    batch = resp.json()['results']
    if not batch:
        break
    all_items.extend(batch)
    page += 1
```

Pagination Loop (R)

```r
all_items <- list()
page <- 1

repeat {
  resp <- GET(url, query = list(page = page))
  batch <- content(resp)$results
  if (length(batch) == 0) break
  all_items <- c(all_items, batch)
  page <- page + 1
}
```

Rate Limits Exist for a Reason

APIs enforce:

- Requests per second
- Requests per day
- Burst limits

Violating limits leads to:

- Errors
- Temporary bans
- Revoked access

Respecting Rate Limits

Good practice includes:

- Sleeping between requests
- Checking headers
- Backing off on errors

```python
import time
time.sleep(1)
```

```r
Sys.sleep(1)
```

Retries and Failures

At scale, failures are normal:

- Network errors
- Timeouts
- Server-side issues

Your code should expect failure.

Retry Pattern (Conceptual)

```python
for attempt in range(3):
    try:
        resp = requests.get(url)
        resp.raise_for_status()
        break
    except Exception:
        time.sleep(2)
```

Logging as Data Collection

Logs help answer:

- What was collected?
- When?
- What failed?
- What was skipped?

Logging is part of reproducibility.

Storing Raw API Responses

Best practice:

- Save raw JSON
- Do not overwrite
- Keep timestamps
- Separate raw from processed data

Raw data are the audit trail.

Data Provenance Matters

Every API workflow should document:

- Endpoint used
- Parameters
- Authentication method
- Collection date
- Known limitations

APIs change over time.

What We Emphasize in Practice

- Treat APIs as institutional artifacts
- Inspect structure before modeling
- Expect scale to surface failure
- Preserve raw data and logs

Discussion

- What assumptions do APIs embed?
- Who controls access and visibility?
- How do rate limits shape research questions?