Data Collection for Machine Learning

Week 1 · CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra

IIT Gandhinagar

Part 1: The Motivation

Why do we need to collect data?

Imagine: You Work at Netflix

> **NETFLIX** — Your Boss: *"We have $500M budget for movie acquisitions. Which movies should we license?"*
>
> **The Question**: Can we predict which movies will succeed?

Your Role: Data Scientist

Your Mission: Build a model to predict movie success

The Problem Statement

Goal: Predict box office revenue based on movie attributes

But wait... What features? What data? Where does it come from?

What We Need: The Target Dataset

Title	Year	Genre	Budget	Revenue	Rating	Director	Cast
Inception	2010	Sci-Fi	$160M	$836M	8.8	C. Nolan	DiCaprio
Avatar	2009	Action	$237M	$2.9B	7.9	Cameron	Worthington
The Room	2003	Drama	$6M	$1.9M	3.9	Wiseau	Wiseau
...	...	...	...	...	...	...	...

We need 10,000+ movies with complete information.

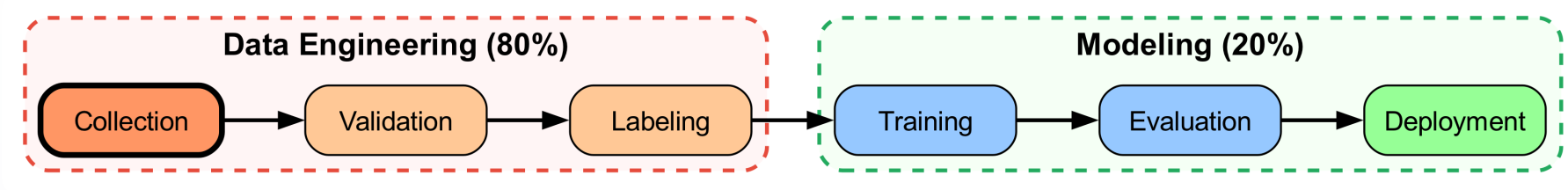
Question: Where does this data come from?

The Reality Check

- This data doesn't exist in one place
- No single CSV file with everything
- Can't just "download" the dataset
- **We must BUILD the dataset ourselves**

This is the real world of data science.

The ML Pipeline Reality



The uncomfortable truth:

- 80% of ML work is data engineering
- Models are the easy part
- **Garbage In = Garbage Out**

Why Is Data Collection So Hard?

The Data Collection Paradox: The data you need rarely exists in the form you need it.

Challenge	Example
Scattered sources	IMDb, Box Office Mojo, Rotten Tomatoes
Different formats	JSON, HTML, CSV
Missing values	Budget missing for 40% of movies
Inconsistent naming	"The Dark Knight" vs "Dark Knight, The"
Rate limits	Only 100 requests/day

Today's Mission

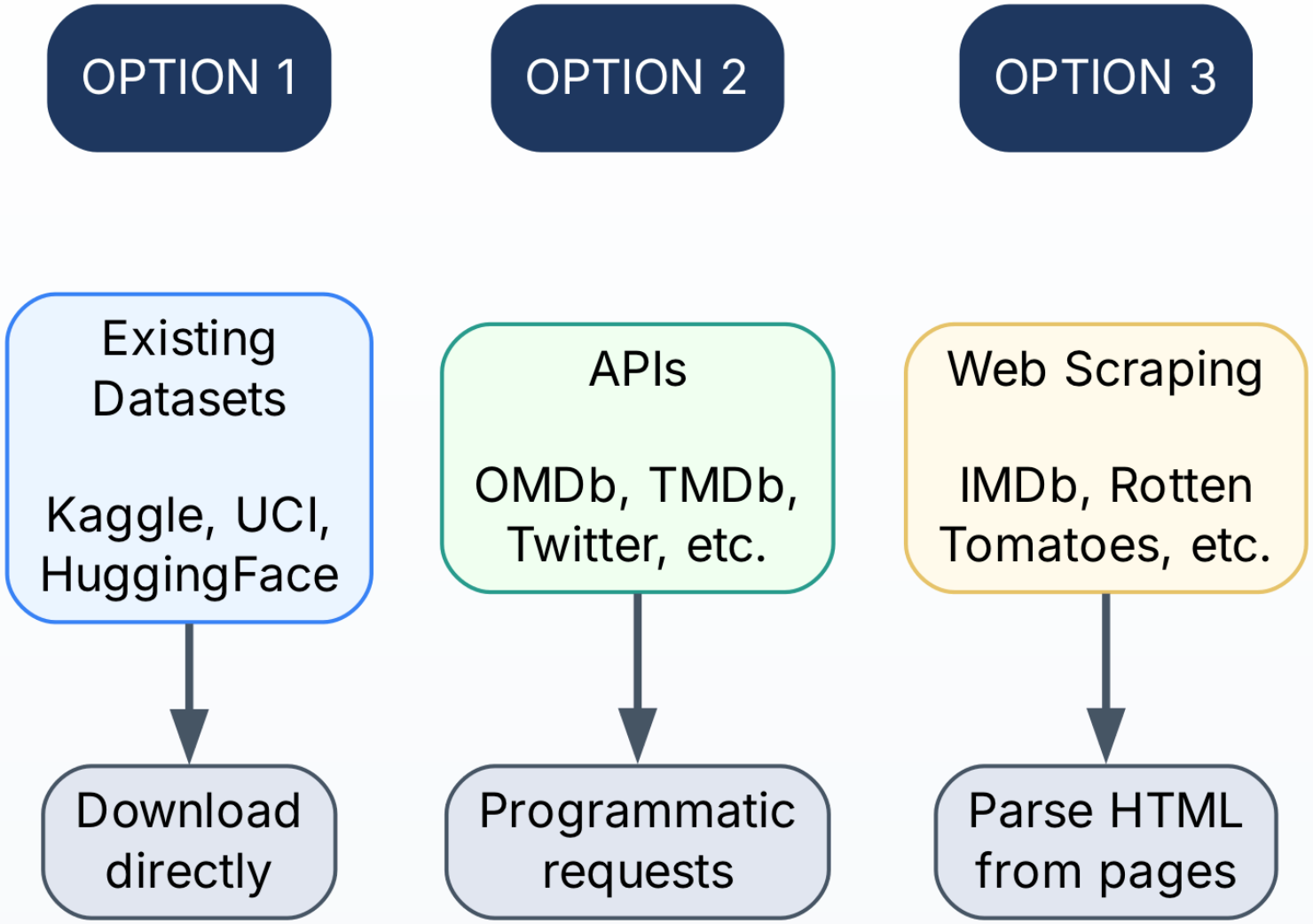
By the end of this lecture, you will know how to:

1. Find data sources for any project
2. Understand how the web works (HTTP)
3. Use Chrome DevTools to inspect network traffic
4. Make requests using curl from the command line
5. Write Python scripts with the requests library
6. Handle different data formats
7. Scrape websites when APIs don't exist

Part 2: Where Does Data Come From?

Finding the right sources

Three Ways to Get Data



Option 1: Pre-built Datasets

Where to find them:

Source	Example Datasets	Pros	Cons
Kaggle	Movies, Titanic, Housing	Ready to use, competitions	May be outdated
UCI ML Repository	Classic ML datasets	Well-documented	Academic focus
HuggingFace	NLP datasets, models	Easy loading	Specialized
Government Portals	Census, economic data	Authoritative	Limited scope

Verdict: Great starting point, but often not enough for real projects.

Option 2: APIs (Application Programming Interface)

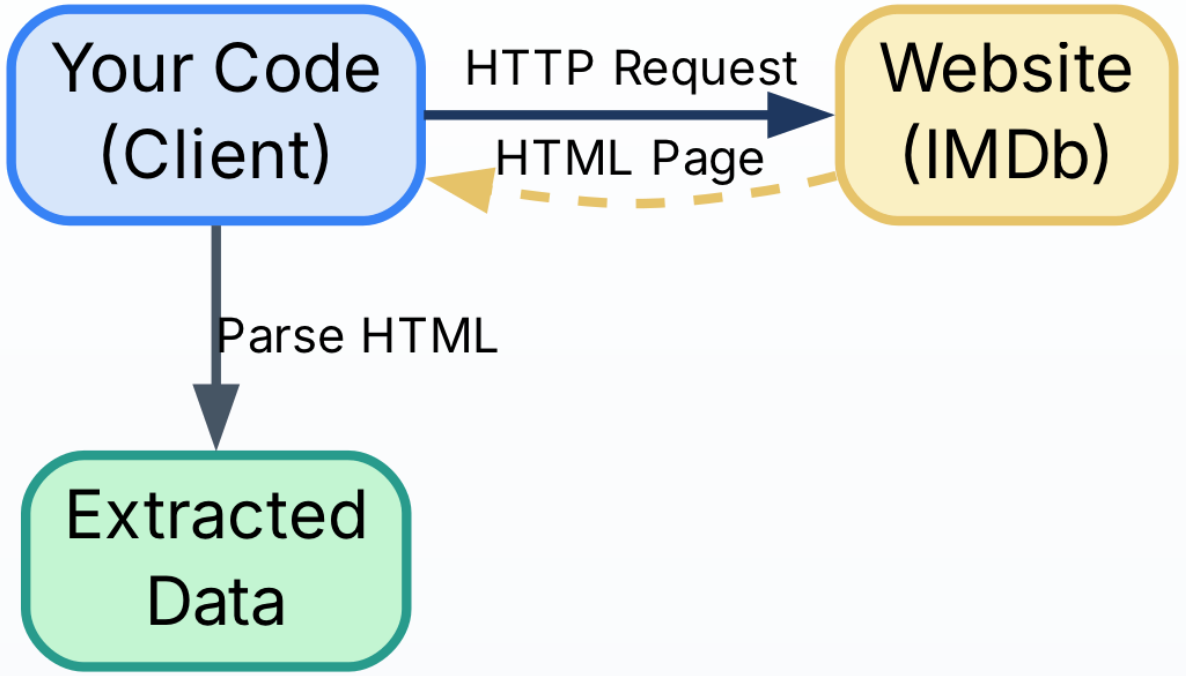
APIs = Structured way to request data from servers

Examples for our Netflix project:

- **OMDb API**: Movie metadata (title, year, ratings)
- **TMDb API**: Detailed movie info, cast, crew
- **Box Office Mojo**: Revenue data

Option 3: Web Scraping

When APIs don't exist or don't have what you need:



When to scrape: Reviews, prices, content not in APIs.

Our Strategy for Netflix Project

Data Needed	Source	Method
Movie titles, years	OMDb API	API calls
Ratings, genres	OMDb API	API calls
Budget, revenue	TMDb API	API calls
User reviews	IMDb website	Scraping
Critic reviews	Rotten Tomatoes	Scraping

Today's focus: Learn both API calls and scraping.

Decision Tree: How to Get Data

Ask these questions in order:

1. Does a ready-made dataset exist?
   → YES: Download it (Kaggle, HuggingFace)
   → NO: Continue to step 2...
2. Does an official API exist?
   → YES: Is it free/affordable? → Use the API
   → NO: Continue to step 3...

Decision Tree (continued)

3. Can you scrape the website?
   - → Check robots.txt and ToS first
   - → YES: Scrape ethically
   - → NO: Look for alternatives

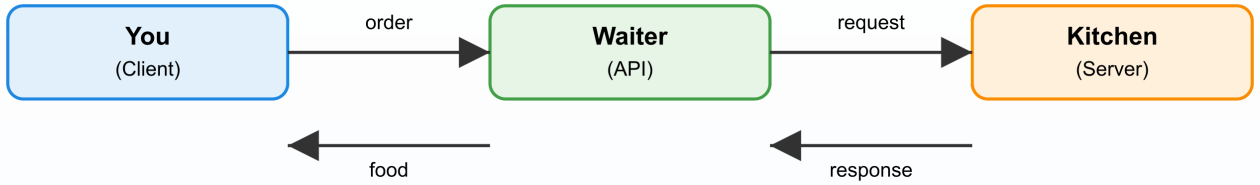
4. None of the above?
   - → Manual data collection
   - → Partner with data owner
   - → Reframe the problem

Most real projects use a combination of all methods!

Part 3: What is an API?

The contract between programs

API: A Restaurant Analogy



Restaurant	API
Menu	Documentation
Order	Request
Kitchen	Server
Food	Response

Our Sample Database

movies.db - SQLite Database

id	title	year	genre	director	rating	budget_millions	revenue_millions
1	Inception	2010	Sci-Fi	Christopher Nolan	8.8	160.0	837.2
2	Avatar	2009	Action	James Cameron	7.9	237.0	2923.7
3	The Matrix	1999	Sci-Fi	Wachowskis	8.7	63.0	467.2
4	The Dark Knight	2008	Action	Christopher Nolan	9.0	185.0	1006.2

Try it yourself: `sqlite3 data/movies.db "SELECT * FROM movies"`

API: The Formal Definition

API (Application Programming Interface)

A defined set of rules and protocols for building and interacting with software applications.

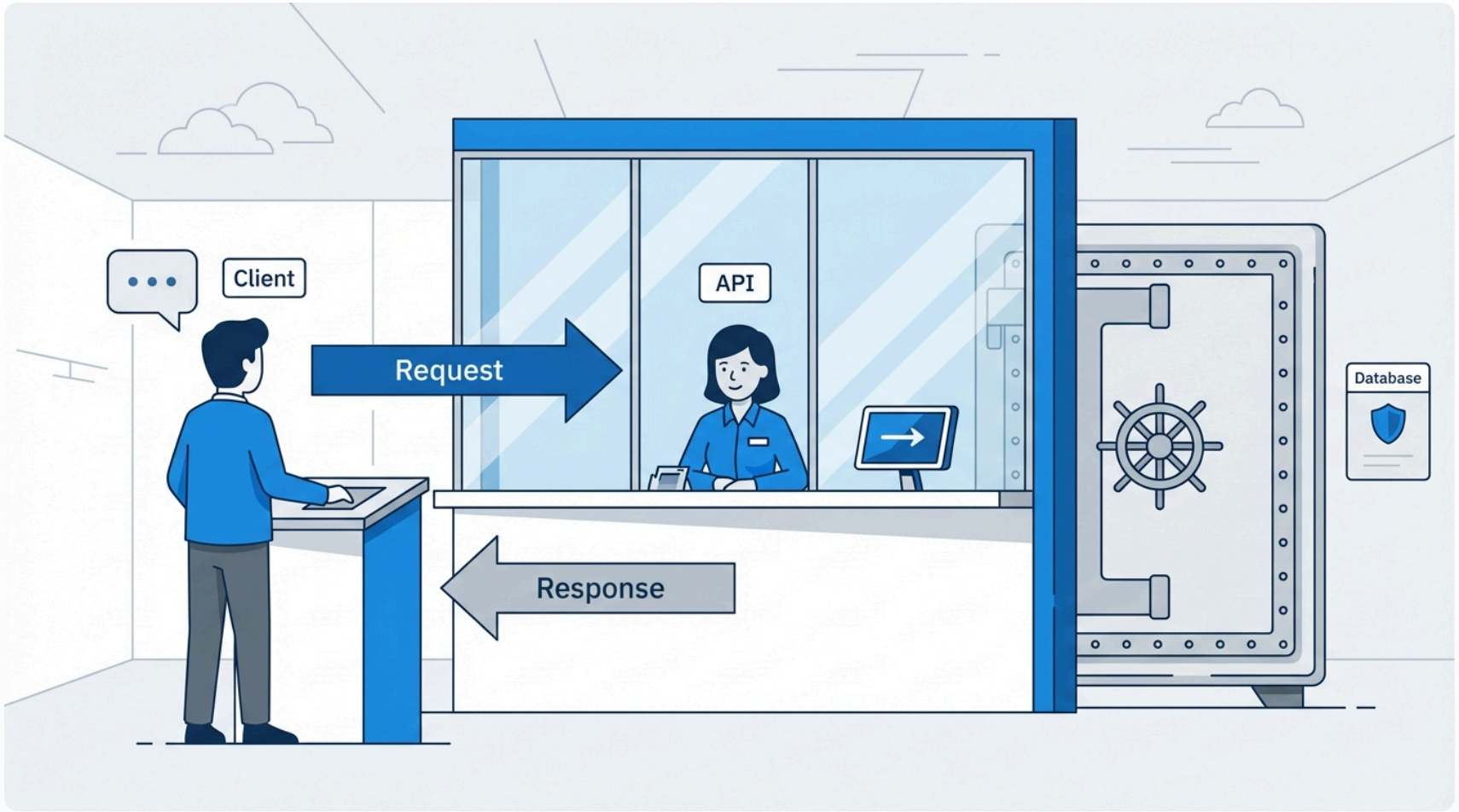
```
# Without API (direct database access - dangerous!)
cursor.execute("SELECT * FROM movies WHERE title = 'Inception'")
# Returns: (1, 'Inception', 2010, 'Sci-Fi', 'Christopher Nolan', 8.8, 160.0, 836.0)

# With API (safe, controlled access)
requests.get("https://nipun-api-testing.hf.space/items")
# Returns: {"items": [...], "count": 3}
```

APIs provide:

- Security (no direct DB access)
- Rate limiting (fair usage)
- Versioning (backwards compatibility)
- Documentation (how to use it)

Why Do APIs Exist?



APIs are like a bank teller window. You can't walk into the vault, but you can request transactions through a controlled interface.

APIs Provide Protection

Without APIs	With APIs
Anyone reads ALL data	Only expose what you want
Anyone can modify/delete	Validate every request
No tracking	Log and monitor usage
Server overwhelmed	Rate limiting protects resources

Reading API Documentation

Before making any API call, check the docs for:

1. **Base URL** - Where do requests go?
2. **Authentication** - API key? Where does it go?
3. **Endpoints** - What resources are available?
4. **Rate limits** - How many requests per day?

Example: OMDb API Docs

```
Base URL: https://www.omdbapi.com/
Auth: apikey parameter in URL
Parameters: t (title), i (IMDb ID), y (year)
Rate limit: 1,000 requests/day (free tier)
```

Get your free API key

Types of APIs

Type	Description	Example
REST API	HTTP-based, stateless, resource-oriented	OMDb, GitHub
GraphQL	Query language, get exactly what you need	GitHub v4, Shopify
SOAP	XML-based, enterprise	Legacy banking
WebSocket	Real-time, bidirectional	Chat apps, live data

For data collection, we focus on REST APIs (most common).

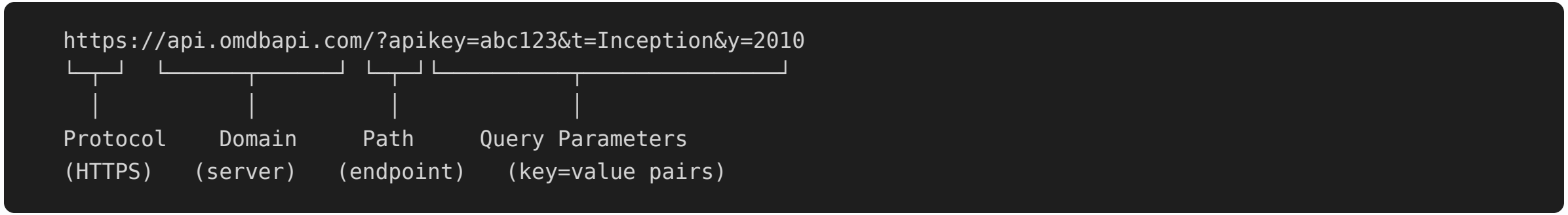
REST API: Key Principles

REST = **RE**presentational **S**tate **T**ransfer

1. **Stateless**: Server doesn't remember previous requests
2. **Resource-based**: URLs represent things (nouns)
3. **HTTP Methods**: Standard verbs (GET, POST, PUT, DELETE)
4. **Standard formats**: JSON or XML responses

```
Good URL Design:
GET  /movies           → List all movies
GET  /movies/123       → Get movie with ID 123
POST /movies           → Create new movie
PUT  /movies/123       → Update movie 123
DELETE /movies/123     → Delete movie 123
```

Anatomy of an API Call



Query Parameters (after the `?`):

- `apikey=abc123` → Authentication
- `t=Inception` → Movie title
- `y=2010` → Year (optional filter)

Multiple parameters joined with `&`

API Authentication

Most APIs require authentication to:

- Track usage
- Enforce rate limits
- Bill customers

Common methods:

```
# 1. API Key in URL (simplest)
GET /movies?apikey=YOUR_KEY

# 2. API Key in Header
GET /movies
X-API-Key: YOUR_KEY

# 3. Bearer Token (OAuth)
GET /movies
Authorization: Bearer YOUR_TOKEN
```

Rate Limiting

Why? Servers have limited resources.

Tier	Requests/Day
Free	100
Basic	1,000
Pro	10,000

If you exceed: HTTP 429 (Too Many Requests)

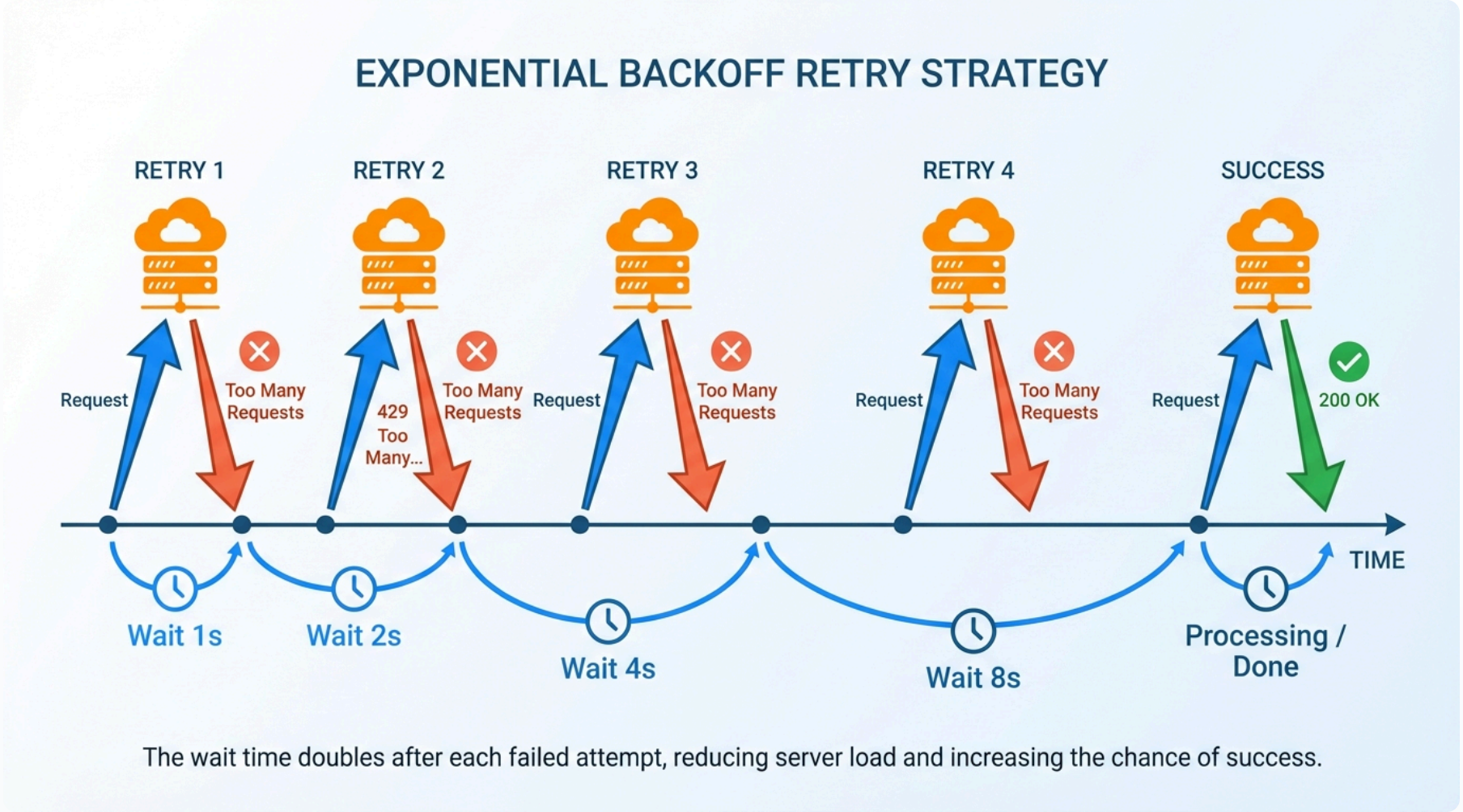
Check headers: `X-RateLimit-Remaining: 42`

Dealing with Rate Limits

Strategy 1: Simple delay

```
for movie in movies:
    response = requests.get(api_url, params={"t": movie})
    time.sleep(1)  # Wait 1 second between requests
```

Exponential Backoff



Wait longer after each failure: 1s → 2s → 4s → 8s → success!

Part 4: HTTP Fundamentals

The language of the web

What is HTTP?

HTTP = **H**yper**T**ext **T**ransfer **P**rotocol

The foundation of data communication on the web.

Key characteristics:

- **Stateless**: Each request is independent
- **Text-based**: Human-readable (mostly)
- **Port 80** (HTTP) or **Port 443** (HTTPS)

Understanding "Stateless"

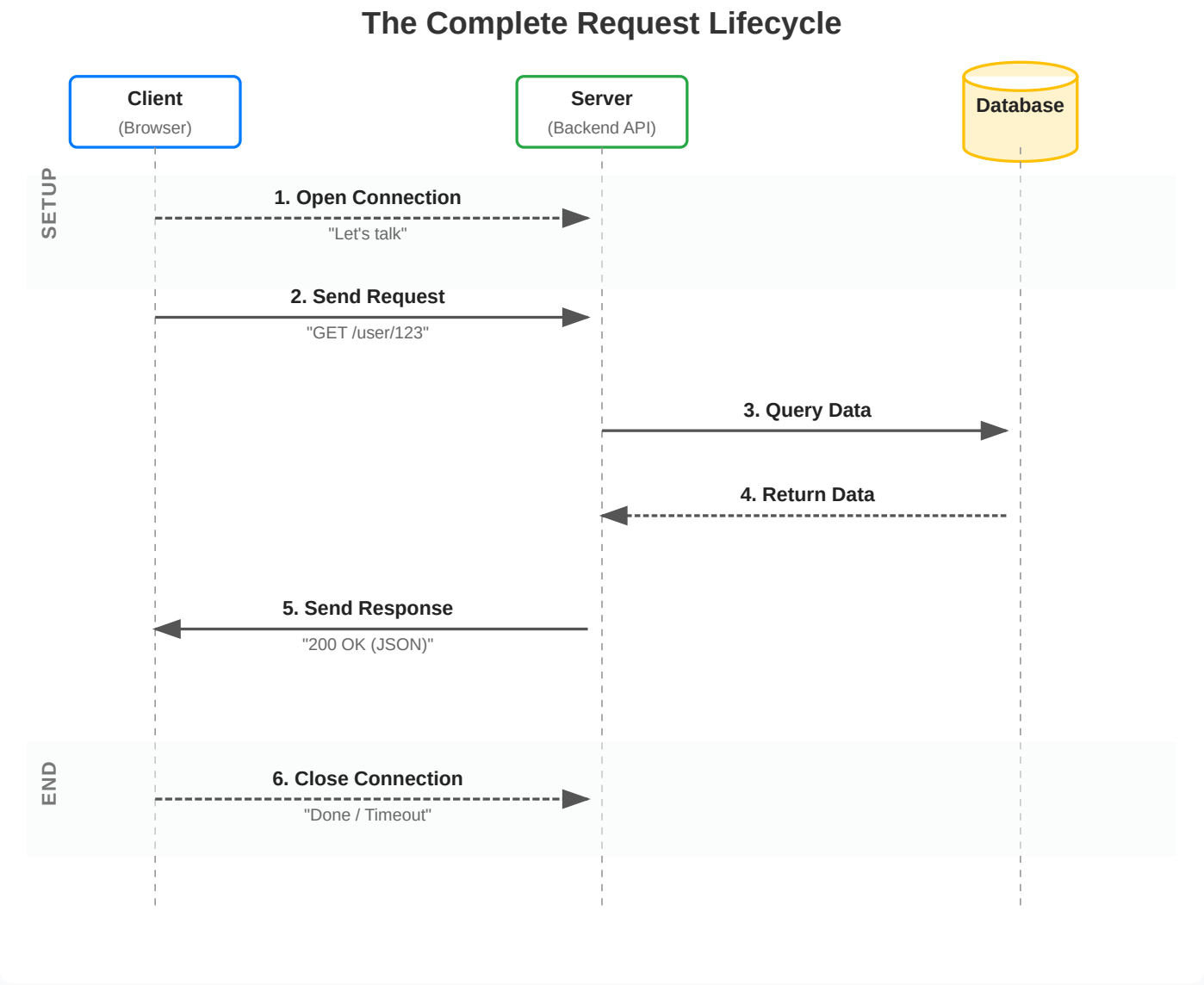
> **The Goldfish Analogy**: Server forgets you after every request.

```
Request 1: "I'm Alice. Show me Inception." → "Here's data."
Request 2: "Now show me Avatar." → "Who are you?"
```

Why stateless? Scalability - any server can handle any request.

Workaround: Cookies, tokens, session IDs (sent with every request)

The Client-Server Model



Chrome DevTools: Your HTTP Inspector

Open DevTools: `F12` or `Cmd+Option+I` (Mac) / `Ctrl+Shift+I` (Windows)

DevTools lets you see:

- Every HTTP request your browser makes
- Request/response headers and body
- **Copy requests as curl commands!**

Navigate to the "Network" tab → This is where the magic happens

Try It: Visit iitgn.ac.in

What your browser sends:

```
GET / HTTP/1.1
Host: iitgn.ac.in
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)...
```

What server responds:

```
HTTP/1.1 200 OK
Server: Apache
Content-Type: text/html; charset=UTF-8
```

Try It: Visit Teaching API

Filter by "Fetch/XHR" to see API calls:

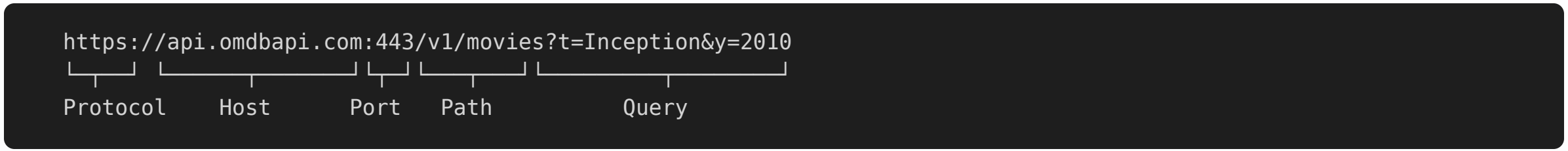
```
GET /items HTTP/1.1
Host: nipun-api-testing.hf.space
Accept: */*
```

Response (JSON):

```
{"items": [{"id": 1, "name": "Apple", "price": 1.5}, ...], "count": 3}
```

Right-click → "Copy as cURL" to replay in terminal!

URL Anatomy



Component	Example
Protocol	`https://` (secure)
Host	`api.omdbapi.com`
Path	`/v1/movies`
Query	`?t=Inception&y=2010`

Key HTTP Headers

Header	What it does
`Host`	Which server to contact
`User-Agent`	Identifies your browser/script
`Accept`	What format you want back
`Content-Type`	Format of data you're sending
`Authorization`	Your API key or token

HTTP Methods: GET vs POST

	GET	POST
Purpose	Retrieve data	Submit data
Parameters	In URL (`?key=value`)	In body
Example	Search, fetch details	Login, upload, create
Data collection	90% of the time	10% of the time

```
# GET - parameters in URL
curl "https://api.example.com/search?q=inception"

# POST - data in body
curl -X POST https://api.example.com/login -d '{"user":"alice"}'
```

HTTP Status Codes

Status codes are grouped by category:

Range	Category	Meaning
1xx	Informational	Request received, processing
2xx	Success	Request succeeded
3xx	Redirection	Further action needed
4xx	Client Error	Your fault
5xx	Server Error	Their fault

Common Status Codes

Code	Meaning	When
`200 OK`	Success	Request succeeded
`201 Created`	Created	POST created resource
`400 Bad Request`	Client error	Malformed request
`401 Unauthorized`	Auth needed	Missing credentials
`403 Forbidden`	Denied	Not allowed
`404 Not Found`	Missing	Resource doesn't exist
`429 Too Many Requests`	Rate limit	Slow down!
`500 Internal Error`	Server crash	Their fault

Status Code Intuition

First digit = who's to blame: 2xx = OK, 4xx = your fault, 5xx = their fault

```
if response.status_code == 200:
    data = response.json()       # Success!
elif response.status_code == 404:
    print("Not found")           # Bad ID
elif response.status_code == 429:
    time.sleep(60)               # Rate limited
elif response.status_code >= 500:
    time.sleep(5)                # Server error
```

Part 6: Response Formats

Same data, different representations

Why Different Formats?

Same movie data can be represented in different formats:

Format	Full Name	Use Case
JSON	JavaScript Object Notation	APIs, Web apps
XML	eXtensible Markup Language	Enterprise, Legacy
CSV	Comma Separated Values	Spreadsheets, ML
HTML	HyperText Markup Language	Web pages
Protobuf	Protocol Buffers	High-performance

Content-Type header tells you the format:

- `application/json` → JSON
- `application/xml` → XML
- `text/html` → HTML
- `text/csv` → CSV

Format 1: JSON

The most common API format today. Try it live:

```
curl https://nipun-api-testing.hf.space/format/json
```

```
{
  "format": "JSON",
  "content_type": "application/json",
  "data": {"name": "Alice", "age": 30, "city": "Mumbai"}
}
```

Pros: Human-readable, lightweight, native to JavaScript

Cons: No schema validation, no comments

JSON Data Types

```json
{
  "string": "Hello World",
  "number": 42,
  "decimal": 3.14159,
  "boolean": true,
  "null_value": null,
  "array": [1, 2, 3],
  "object": {
    "nested": "value"
  }
}
```

Only 7 data types: string, number, boolean, null, array, object

Note: No native date type! Dates are typically strings: `"2010-07-16"`

JSON Gotchas

```
# Numbers might be strings!
data = {"year": "2010"}     # String, not int!
year = int(data["year"])    # Must convert

# Missing keys crash your code
data["director"]                # KeyError!
data.get("director", "Unknown") # Safe!
```

More JSON Gotchas

```
# null becomes None in Python
data = {"budget": None}
if data["budget"]:        # This is False!
    print("Has budget")

# Empty string vs null vs missing
{"rating": ""}      # Empty string
{"rating": None}    # Null
{}                  # Missing key
```

Format 2: XML

The enterprise standard (still used in SOAP APIs). Try it live:

```
curl https://nipun-api-testing.hf.space/format/xml
```

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
    <format>XML</format>
    <data>
        <user>
            <name>Alice</name>
            <age>30</age>
            <city>Mumbai</city>
        </user>
    </data>
</response>
```

Pros: Schema validation (XSD), attributes, widespread support

Cons: Verbose, heavier than JSON

JSON vs XML: Same Data

Aspect	JSON	XML
Syntax	`{"name": "Inception"}`	`<name>Inception</name>`
Structure	Curly braces `{}`	Tags `<tag></tag>`
Size	Lighter (~30% smaller)	More verbose
Attributes	Not supported	Supported
Arrays	`[1, 2, 3]`	Repeated elements
Usage	Modern APIs	Legacy/Enterprise

Format 3: CSV

The data scientist's friend. Try it live:

```
curl https://nipun-api-testing.hf.space/format/csv
```

```
id,name,price,quantity,description
1,Apple,1.50,100,Fresh red apple
2,Banana,0.75,150,Yellow banana
3,Orange,2.00,80,Juicy orange
```

Pros: Opens in Excel, `pd.read_csv()`, very compact

Cons: Flat structure only, no data types, escaping issues

Format 4: HTML

What you get when scraping websites.

```
<div class="movie-card">
  <h2 class="title">Inception</h2>
  <span class="year">2010</span>
  <ul class="genres">
    <li>Sci-Fi</li>
    <li>Action</li>
  </ul>
  <p class="rating">Rating: 8.8/10</p>
</div>
```

Not designed for data exchange!

- Mixed with presentation (CSS, layout)
- Need to parse and extract relevant data
- Structure varies by website

Format 5: Protocol Buffers (Protobuf)

Google's high-performance binary format.

```
// movie.proto (schema definition)
message Movie {
  string title = 1;
  int32 year = 2;
  repeated string genres = 3;
  float rating = 4;
}
```

```
# After compiling: protoc --python_out=. movie.proto
from movie_pb2 import Movie
movie = Movie(title="Inception", year=2010, genres=["Sci-Fi", "Action"], rating=8.8)
binary_data = movie.SerializeToString()  # Only 25 bytes!
print(binary_data.hex())  # 0a09496e63657074696f6e10da0f...
```

Pros: 10x smaller, 100x faster parsing

Cons: Need schema, binary format, requires tooling

Format Comparison: Same Movie

Format	Size	Readability	Use Case
JSON	150 bytes	High	REST APIs
XML	200 bytes	Medium	Enterprise
CSV	50 bytes	High	Data exchange
HTML	300 bytes	Low	Web pages
Protobuf	30 bytes	None	High-perf APIs

For this course: Focus on JSON and HTML

Part 7: Making Requests with curl

The command-line HTTP client

What is curl?

curl = "Client URL" - a command-line tool for transferring data.

```
# Your first curl command
curl "https://www.omdbapi.com/?t=Inception&apikey=[API_KEY]"
```

Why learn curl?

- Universal (works everywhere)
- Quick debugging
- Foundation for understanding HTTP
- Copy from DevTools, paste and run

curl: Basic Syntax

```
curl [options] [URL]
```

Common options:

Option	Meaning	Example
`-X`	HTTP method	`-X POST`
`-H`	Add header	`-H "Accept: application/json"`
`-d`	Send data (body)	`-d '{"key": "value"}'`
`-o`	Output to file	`-o movie.json`
`-I`	Headers only	`-I`
`-v`	Verbose output	`-v`
`-s`	Silent mode	`-s`

curl: GET Request

```
# Try these right now! (no API key needed)
curl https://nipun-api-testing.hf.space/hello
# {"message": "Hello, World!"}

curl https://nipun-api-testing.hf.space/items
# {"items": [{"id": 1, "name": "Apple", ...}], "count": 3}

curl "https://nipun-api-testing.hf.space/greet?name=Alice"
# {"greeting": "Hello, Alice!"}
```

Important: Quote URLs with `?` or `&` (prevents shell interpretation)

curl: Real API Example (OMDb)

For actual movie data, use OMDb API (free tier: 1000 requests/day)

```
# Get movie by title (requires API key)
curl "https://www.omdbapi.com/?t=Inception&apikey=YOUR_KEY"
```

```
{
  "Title": "Inception", "Year": "2010", "Rated": "PG-13",
  "Genre": "Action, Adventure, Sci-Fi",
  "Director": "Christopher Nolan",
  "imdbRating": "8.8", "imdbID": "tt1375666"
}
```

Get your free key: https://www.omdbapi.com/apikey.aspx

curl: Adding Headers

```
curl "https://www.omdbapi.com/?t=Inception&apikey=[API_KEY]" \
     -H "Accept: application/json" \
     -H "Authorization: Bearer YOUR_TOKEN" \
     -H "User-Agent: MyApp/1.0"
```

Common headers to add:

- `Accept: application/json` - Request JSON response
- `Authorization: Bearer TOKEN` - Authentication
- `Content-Type: application/json` - When sending JSON

curl: Viewing Response Headers

```
# Show only response headers (no body)
curl -I "https://www.omdbapi.com/?t=Inception&apikey=[API_KEY]"
```

Output:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 1024
Cache-Control: public, max-age=86400
X-RateLimit-Remaining: 999
```

curl: Verbose Mode

```
curl -v "https://www.omdbapi.com/?t=Inception&apikey=[API_KEY]"
```

Shows everything (request AND response):

```
> GET /?apikey=demo&t=Inception HTTP/2
> Host: api.omdbapi.com
> User-Agent: curl/7.79.1
> Accept: */*
>
< content-length: 1024
<
{"Title":"Inception"...}
```

`>` = What you sent (request)

`<` = What you received (response)

Pretty Printing with jq

Raw JSON is hard to read. Pipe to `jq` for formatting:

```
curl -s https://nipun-api-testing.hf.space/items | jq .
```

```
{"items": [{"id": 1, "name": "Apple", ...}, ...], "count": 3}
```

jq: Extracting and Transforming Data

```
# Get just the items array
curl -s https://nipun-api-testing.hf.space/items | jq '.items'

# Get first item only
curl -s ... | jq '.items[0]'
# {"id": 1, "name": "Apple", "price": 1.5, ...}

# Get all names
curl -s ... | jq '.items[].name'
# "Apple"  "Banana"  "Orange"

# Create new structure
curl -s ... | jq '.items[] | {product: .name, cost: .price}'
```

More on jq next week!

curl: Saving to File

```
# Save response to file
curl "https://www.omdbapi.com/?t=Inception&apikey=[API_KEY]" \
     -o inception.json

# Silent mode (no progress bar)
curl -s "https://www.omdbapi.com/?t=Inception&apikey=[API_KEY]" -o output.json
# Save with pretty formatting
curl -s ... | jq . > formatted.json
```

curl: POST Request

```
curl -X 'POST' \
  'https://nipun-api-testing.hf.space/items' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "name": "Laptop",
  "price": 999.99,
  "quantity": 1,
  "description": "A powerful laptop"
}'
```

Components:

- `-X POST` - Use POST method
- `-H "Content-Type: application/json"` - Tell server we're sending JSON
- `-d '...'` - The data (request body)

curl: POST with Form Data

```
curl -X POST "https://nipun-api-testing.hf.space/form/contact" \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "name=Alice" \
  -d "email=alice@example.com" \
  -d "subject=Hello" \
  -d "message=Nice API!"
```

curl: File Upload

```
# Upload a file
curl -X POST "https://nipun-api-testing.hf.space/upload/file" -F "file=@dummy.txt"
```

`-F` = multipart form data (for file uploads)
`@` = read from file

curl: Useful Options

```
# Retry on failure
curl --retry 3 "https://www.omdbapi.com/data"

# Set timeout (seconds)
curl --max-time 10 "https://www.omdbapi.com/slow"

# Follow redirects
curl -L "https://short.url/abc"

# Fail silently on HTTP errors
curl -f "https://www.omdbapi.com/notfound"
# (exits with error code instead of showing error page)
```

Part 9: Python requests Library

Programmatic data collection

Why Python requests?

curl is great for testing, but for automation you need Python.

```
# Install
pip install requests
```

Benefits over curl:

- Loop over many URLs
- Parse JSON automatically
- Handle errors gracefully
- Store data in variables
- Integrate with pandas, ML pipelines

requests: Simple GET

```
import requests

# Make a GET request to OMDb API
response = requests.get(
    "https://www.omdbapi.com/",
    params={
        "apikey": "demo",   # replace with your real API key
        "t": "Inception"
    }
)

# Check HTTP status code
print(response.status_code)  # 200 means OK

# Parse JSON response
data = response.json()
```

requests: Using params

Don't manually build query strings!

```
# Bad (manual string building)
url = "https://www.omdbapi.com/?apikey=demo&t=Inception&y=2010"

# Good (use params dict)
response = requests.get(
    "https://www.omdbapi.com/",
    params={
        "apikey": "demo",
        "t": "Inception",
        "y": 2010
    }
)
```

Python handles URL encoding automatically!

requests: Adding Headers

```
import requests

response = requests.get(
    "https://httpbin.org/headers",
    headers={
        "Authorization": "Bearer test-token-123",
        "Accept": "application/json",
        "User-Agent": "MyApp/1.0"
    }
)

print(response.status_code)
print(response.json())
```

requests: Response Object

```python
import requests
response = requests.get("https://nipun-api-testing.hf.space/items")

response.status_code        # 200
response.headers["Content-Type"]  # 'application/json'
response.text               # Raw text (string)
response.json()             # Parsed as Python dict
response.ok                 # True for 2xx status codes
```

```
# Example output
>>> response.json()
{'items': [{'id': 1, 'name': 'Apple', ...}], 'count': 3}
```

requests: POST with JSON

```python
import requests

response = requests.post(
    "https://nipun-api-testing.hf.space/items",
    json={"name": "Laptop", "price": 999.99, "quantity": 1}
)
print(response.status_code)  # 201 (Created)
print(response.json())       # {'id': 4, 'name': 'Laptop', ...}
```

requests: POST with Form Data

```
response = requests.post(
    "https://nipun-api-testing.hf.space/form/contact",
    data={"name": "Alice", "email": "alice@example.com", "message": "Hello!"}
)
print(response.json())  # {'status': 'received', 'name': 'Alice', ...}
```

Remember:

- `json=` → sends JSON (Content-Type: application/json)
- `data=` → sends form data (Content-Type: application/x-www-form-urlencoded)

requests: Error Handling

```
try:
    response = requests.get("https://nipun-api-testing.hf.space/items", timeout=10)
    response.raise_for_status()  # Raises exception for 4xx/5xx
    data = response.json()
except requests.exceptions.Timeout:
    print("Request timed out")
except requests.exceptions.HTTPError as e:
    print(f"HTTP error: {e}")
except requests.exceptions.RequestException as e:
    print(f"Request failed: {e}")
```

Key points:

- Always set `timeout` to avoid hanging forever
- `raise_for_status()` converts bad status codes to exceptions

requests: Looping Over Multiple Items

```
movies = ["Inception", "Avatar", "The Matrix"]
results = []

for title in movies:
    response = requests.get(
        "https://www.omdbapi.com/",
        params={"apikey": "YOUR_KEY", "t": title}, timeout=10
    )
    if response.ok and response.json().get("Response") == "True":
        results.append(response.json())
        print(f"Got: {title}")
    time.sleep(1)  # Be polite - don't hammer the server

print(f"Collected {len(results)} movies")
```

requests: Practical Example

```
def fetch_movies(titles, api_key):
    movies = []
    for title in titles:
        r = requests.get("https://www.omdbapi.com/",
                         params={"apikey": api_key, "t": title}, timeout=10)
        if r.ok and r.json().get("Response") == "True":
            movies.append(r.json())
        time.sleep(0.5)
    return pd.DataFrame(movies)

df = fetch_movies(["Inception", "Avatar", "The Matrix"], "YOUR_KEY")
print(df[["Title", "Year", "Genre", "imdbRating"]])
```

Data Collection Best Practices

1. **Save raw responses** - Save the full JSON, not just extracted fields
2. **Log everything** - Track successes, failures, and why
3. **Use checkpoints** - Resume after crashes
4. **Handle edge cases** - Missing budgets, directors, etc.
5. **Validate as you go** - Check data types early

Why? Don't re-collect 10,000 movies because you missed a field!

curl vs requests: Comparison

Aspect	curl	Python requests
Use case	Quick testing	Automation
Learning	Interactive exploration	Production code
Looping	Bash scripts	Native Python
JSON parsing	Needs jq	Built-in .json()
Error handling	Exit codes	Exceptions
DevTools	Copy as curl (yes)	Convert from curl

Workflow: DevTools → Copy as curl → Test → Convert to Python

Part 10: Web Scraping

When APIs don't exist

When to Scrape?

DO scrape when:

- No API available
- API doesn't have the data you need
- API is too expensive
- Public information on public websites

DON'T scrape when:

- robots.txt disallows it
- Terms of Service prohibit it
- Data is behind login (personal data)
- It would harm the website

API vs Scraping Comparison

Aspect	API	Scraping
Reliability	Stable	Fragile (HTML changes)
Speed	Fast	Slower
Data Format	Structured JSON	Unstructured HTML
Rate Limits	Documented	Unknown
Legality	Clear TOS	Gray area
Maintenance	Low	High

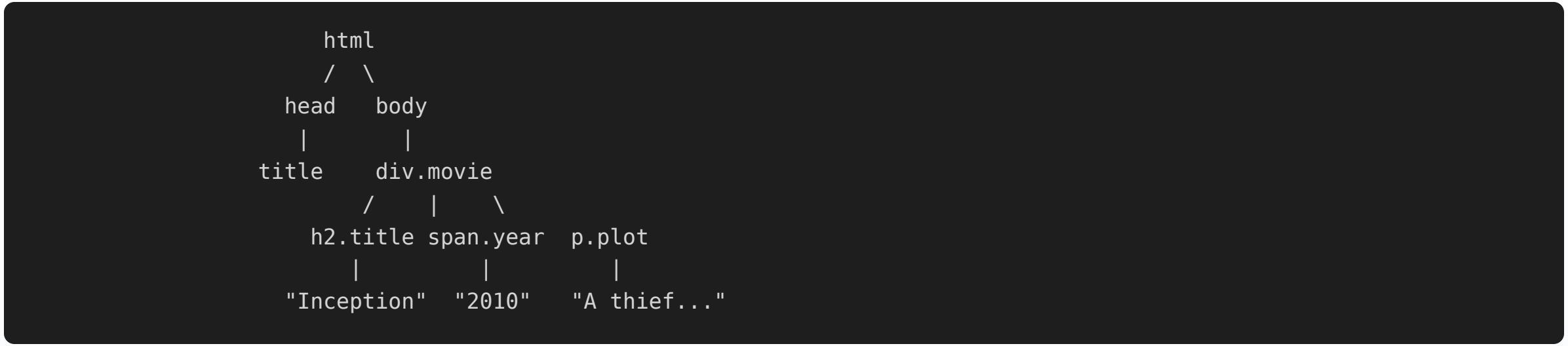
Rule: Always prefer APIs when available.

HTML Structure Basics

HTML = Nested elements forming a tree (DOM)

```
<!DOCTYPE html>
<html>
  <head>
    <title>Movie Database</title>
  </head>
  <body>
    <div class="movie" id="movie-123">
      <h2 class="title">Inception</h2>
      <span class="year">2010</span>
      <p class="plot">A thief who steals...</p>
    </div>
  </body>
</html>
```

The DOM Tree



DOM = Document Object Model

Scraping = Navigating this tree to extract data

CSS Selectors: Finding Elements

Selector	Meaning	Example Match
`div`	Element type	`<div>...</div>`
`.movie`	Class name	`<div class="movie">`
`#main`	Element ID	`<div id="main">`
`div.movie`	Tag with class	`<div class="movie">`
`.movie .title`	Nested element	`.title` inside `.movie`
`a[href="/movies"]`	Attribute value	`<a href="/movies">`

BeautifulSoup: Setup

```
pip install beautifulsoup4 requests
```

```
# Fetch the hosted sample movie page
url = "https://nipunbatra.github.io/stt-ai-teaching/html/sample-movie-website.html"
response = requests.get(url)
html = response.text

# Parse it
soup = BeautifulSoup(html, 'html.parser')

# Now we can search and extract elements
print(soup.title.string)  # "My Movie Library"
```

BeautifulSoup: Finding Elements

```
html = """
<div class="movie">
    <h2 class="title">Inception</h2>
    <span class="year">2010</span>
    <span class="rating">8.8</span>
</div>
"""

soup = BeautifulSoup(html, 'html.parser')

# Find single element
title = soup.find('h2', class_='title')
print(title.text)  # "Inception"

all_movies = soup.find_all('div', class_='movie') # Find all elements (if multiple movies)
```

BeautifulSoup: CSS Selectors

```
soup = BeautifulSoup(html, 'html.parser')

# Select first match
title = soup.select_one('.movie .title')
print(title.text)  # "Inception"

# Select all matches
all_titles = soup.select('.movie .title')
for t in all_titles:
    print(t.text)

# Example: all links starting with "/movies/"
links = soup.select('a[href^="/movies/"]')
for link in links:
    print(link.get('href'))
```

BeautifulSoup: Extracting Data

```
# Get text content
element = soup.select_one('.title')
print(element.text)           # "Inception"
print(element.get_text())     # "Inception"
print(element.get_text(strip=True))  # Remove extra whitespace

# Get attributes
link = soup.select_one('a')
print(link.get('href'))       # "/movies/123"
print(link['href'])           # "/movies/123"
print(link.attrs)             # {'href': '/movies/123', 'class': ['btn']}
```

Scraping Example: Movie List

```
# Scraping Example: Movie List

url = "https://nipunbatra.github.io/stt-ai-teaching/html/sample-movie-website.html"
response = requests.get(url)
soup = BeautifulSoup(response.text, 'html.parser')
movies = []

for card in soup.select('.movie-card'):
    movie = {
        'title': card.select_one('.title').text.strip(),
        'year': card.select_one('.year').text.strip(),
        'genre': card.select_one('.genre').text.strip(),
        'rating': card.select_one('.rating').text.strip(),
        'plot': card.select_one('.plot').text.strip()
    }
    movies.append(movie)
```

Scraping Ethics & Best Practices

```
headers = {'User-Agent': 'MyBot/1.0 (contact@example.com)'}

for url in urls:
    response = requests.get(url, headers=headers)
    time.sleep(1)  # Wait between requests
```

Rules:

1. Check `robots.txt` first
2. Add delays between requests
3. Identify yourself (User-Agent)
4. Cache responses when possible
5. Respect rate limits

Common Scraping Mistakes

Mistake	Solution
No delays	Add `time.sleep(1)`
Hardcoded selectors	Handle missing elements
No error handling	Wrap in try/except
Ignoring encoding	Check `response.encoding`
Not saving raw HTML	Save before parsing

Defensive Scraping Pattern

```
try:
    title = card.select_one('.title')
    movie['title'] = title.text.strip() if title else "Unknown"
except Exception as e:
    logging.error(f"Failed to parse: {url}, error: {e}")
```

> Always handle missing elements gracefully!

Checking robots.txt - Real Examples

```
curl https://www.google.com/robots.txt
```

```
User-agent: *
Disallow: /search        # Can't scrape search results
Allow: /search/about     # But info pages are OK
Disallow: /?             # No query parameters
```

```
curl https://www.amazon.com/robots.txt
```

```
User-agent: *
Disallow: /gp/cart       # No shopping carts
Disallow: /gp/sign-in    # No login pages
Disallow: /gp/yourstore  # No personalized pages
```

Always check before scraping!

Part 11: Putting It All Together

Back to our Netflix mission

Remember Our Goal?

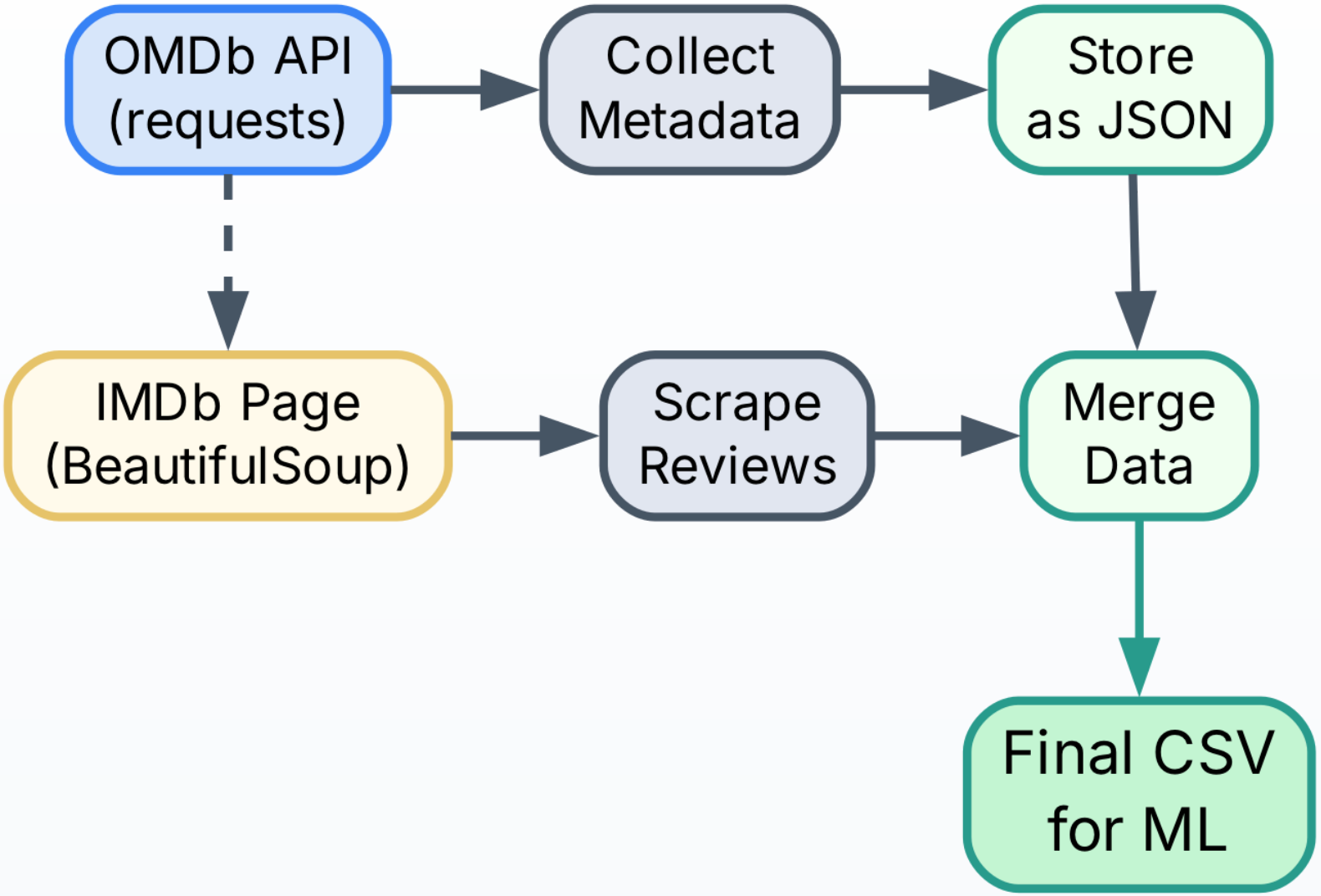
Build a dataset for movie success prediction:

Title	Year	Genre	Budget	Revenue	Rating	Director
?	?	?	?	?	?	?

We now have the tools!

- **DevTools** to find APIs
- **curl** to test requests
- **requests** to automate collection
- **BeautifulSoup** for scraping

Our Data Collection Pipeline



Step 1: Collect from API

```
API_KEY = "your_omdb_key"  # Replace with your actual OMDb API key
movies_to_fetch = ["Inception", "Avatar", "The Matrix"]
results = []

for title in movies_to_fetch:
    response = requests.get(
        "https://www.omdbapi.com/",
        params={"apikey": API_KEY, "t": title},
        timeout=10  # Prevent hanging requests
    )

    if response.ok: # Check HTTP-level success
        data = response.json()

        # Check API-level success
        if data.get("Response") == "True":
            results.append(data)
            print(f"Fetched: {title}")
        else:
```

Step 2: Extract Relevant Fields

```
movies = []

for data in results:
    movie = {
        "title": data.get("Title"),
        "year": data.get("Year"),
        "genre": data.get("Genre"),
        "director": data.get("Director"),
        "rating": data.get("imdbRating"),
        "votes": data.get("imdbVotes"),
        "runtime": data.get("Runtime"),
        "imdb_id": data.get("imdbID")
    }
    movies.append(movie)
```

Step 3: Save to CSV

```
import pandas as pd

# Convert to DataFrame
df = pd.DataFrame(movies)

# Clean data
df['year'] = pd.to_numeric(df['year'], errors='coerce')
df['rating'] = pd.to_numeric(df['rating'], errors='coerce')
df['votes'] = df['votes'].str.replace(',', '').astype(float)

# Save
df.to_csv('netflix_movie_data.csv', index=False)

print(df.head())
```

The Result

```
          title  year                  genre           director  rating
0     Inception  2010  Action, Adventure...  Christopher Nolan     8.8
1        Avatar  2009  Action, Adventure...       James Cameron     7.9
2    The Matrix  1999  Action, Sci-Fi        Lana Wachowski...     8.7
```

Now ready for ML modeling!

What We Learned: Three Tools

Tool	When to Use	Key Commands
Chrome DevTools	Discover APIs, inspect requests	Network tab, Copy as curl
curl	Test requests quickly	`-X`, `-H`, `-d`, `
Python requests	Automate collection	`.get()`, `.post()`, `.json()`

Plus BeautifulSoup for scraping when needed!

Part 12: Looking Ahead

Lab preview and next week

This Week's Lab

Hands-on Practice:

1. **Chrome DevTools** - Inspect API calls on real websites
2. **curl exercises** - Making API requests from terminal
3. **OMDb API** - Collecting movie metadata
4. **Python requests** - Building a data collection script
5. **BeautifulSoup** - Scraping a sample website

Goal: Build a working data collection pipeline.

Lab Environment Setup

```
# Install dependencies
pip install requests beautifulsoup4 pandas

# Get your API keys
# OMDb: https://www.omdbapi.com/apikey.aspx (free tier)

# Verify installation
python -c "import requests; print('Ready!')"
```

Next Week Preview

Week 2: Data Validation & Cleaning

- Schema validation with Pydantic
- Handling missing data
- Type conversion and normalization
- Data quality checks
- Building validation pipelines

The data we collect today needs cleaning tomorrow!

Key Takeaways

1. **Data collection is 80% of ML work** - don't underestimate it
2. **DevTools reveals hidden APIs** - always check before scraping
3. **curl for quick testing** - then convert to Python
4. **requests for automation** - handle loops, errors, storage
5. **Scraping is plan B** - use when APIs don't exist
6. **Be ethical** - respect robots.txt, rate limits, ToS

Resources

Documentation:

- curl - Command-line HTTP client
- requests - Python HTTP library
- BeautifulSoup - HTML parsing

Free APIs for Practice:

- JSONPlaceholder - Fake REST API
- OMDb API - Movie database
- Public APIs - Curated list
- Teaching API - No key needed!

Questions?

Thank You!

See you in the lab!