Week 1: Data Collection for Machine Learning

CS 203: Software Tools and Techniques for AI

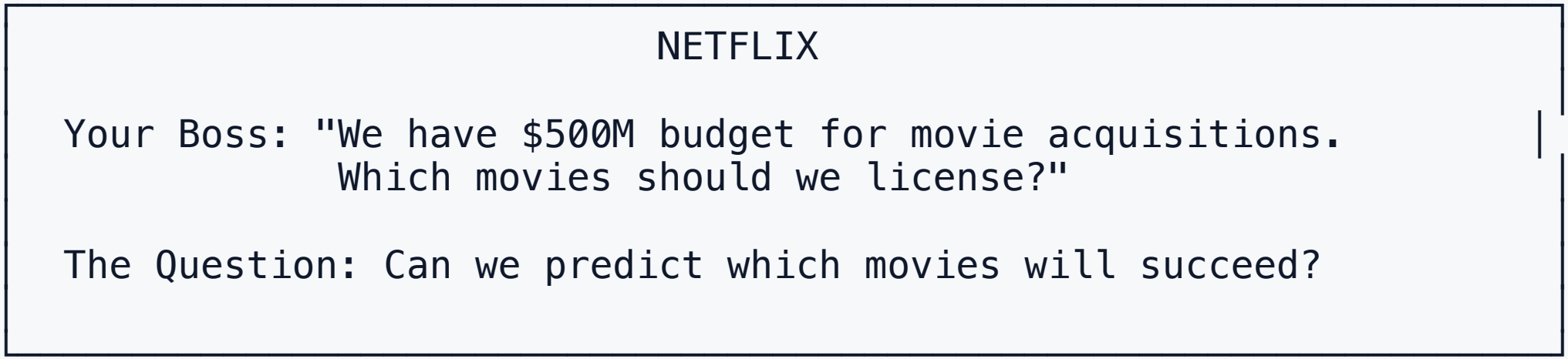
Prof. Nipun Batra

IIT Gandhinagar

Part 1: The Motivation

Why do we need to collect data?

Imagine: You Work at Netflix



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

```
[X] This data doesn't exist in one place

[X] No single CSV file with everything

[X] 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**

Today's Mission

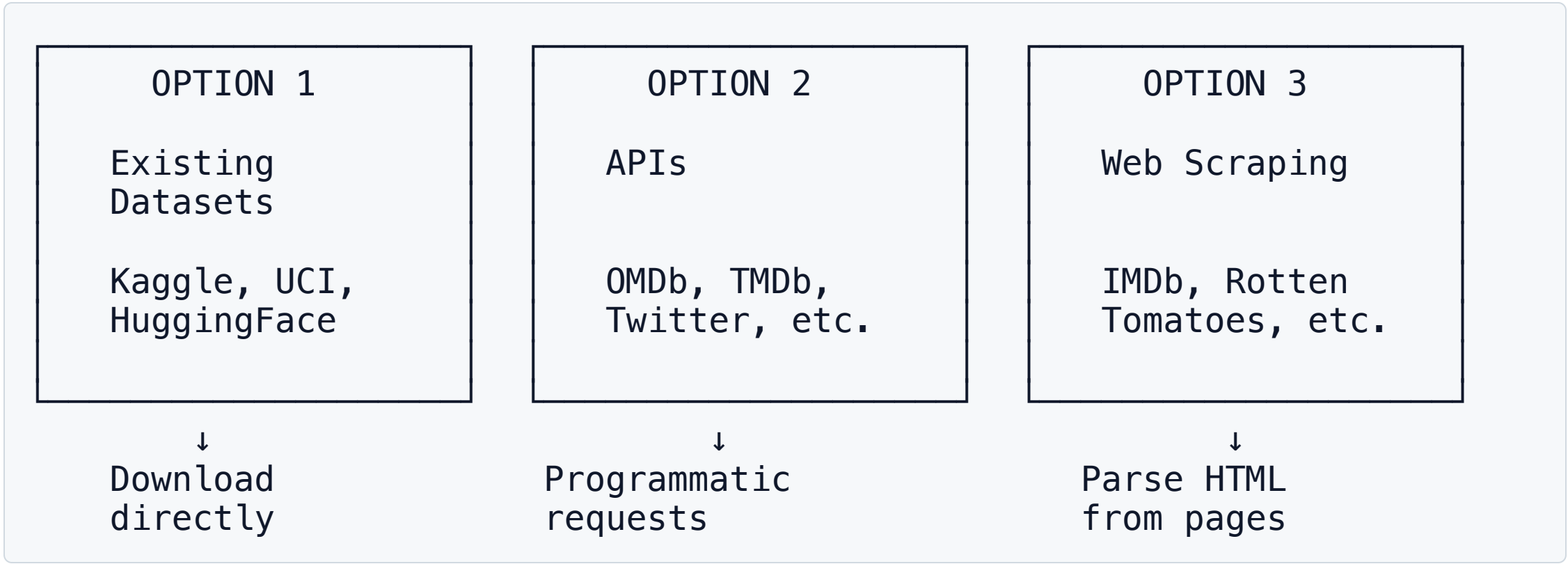
By the end of this lecture, you will know how to:

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



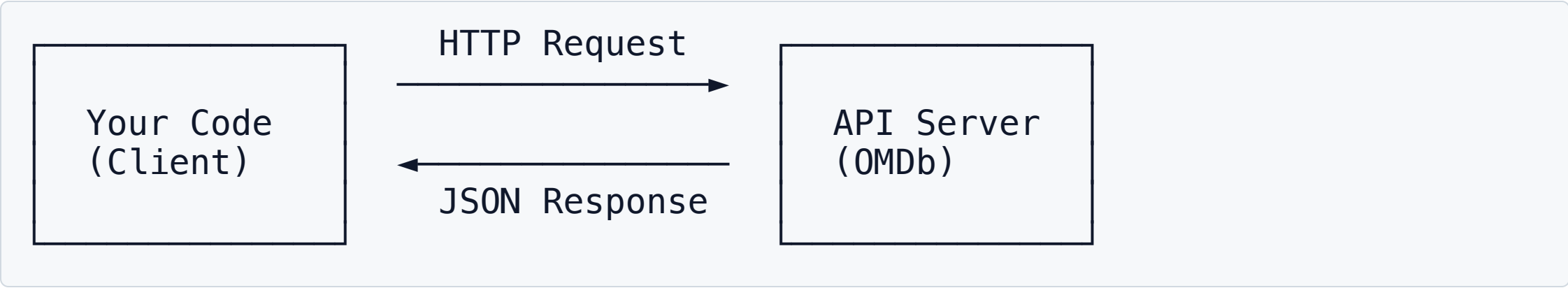
Option 1: Pre-built Datasets

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)



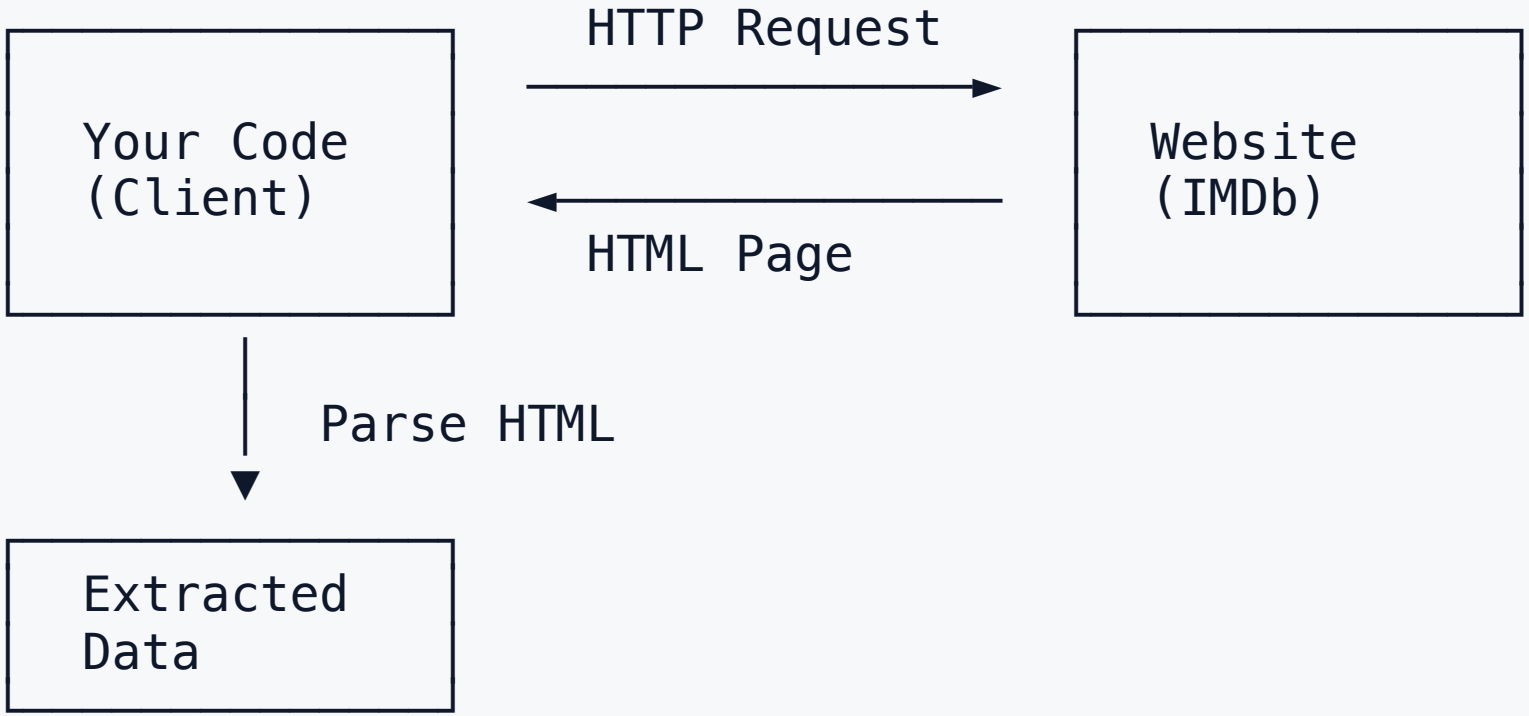
APIs = Structured way to request data from servers

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

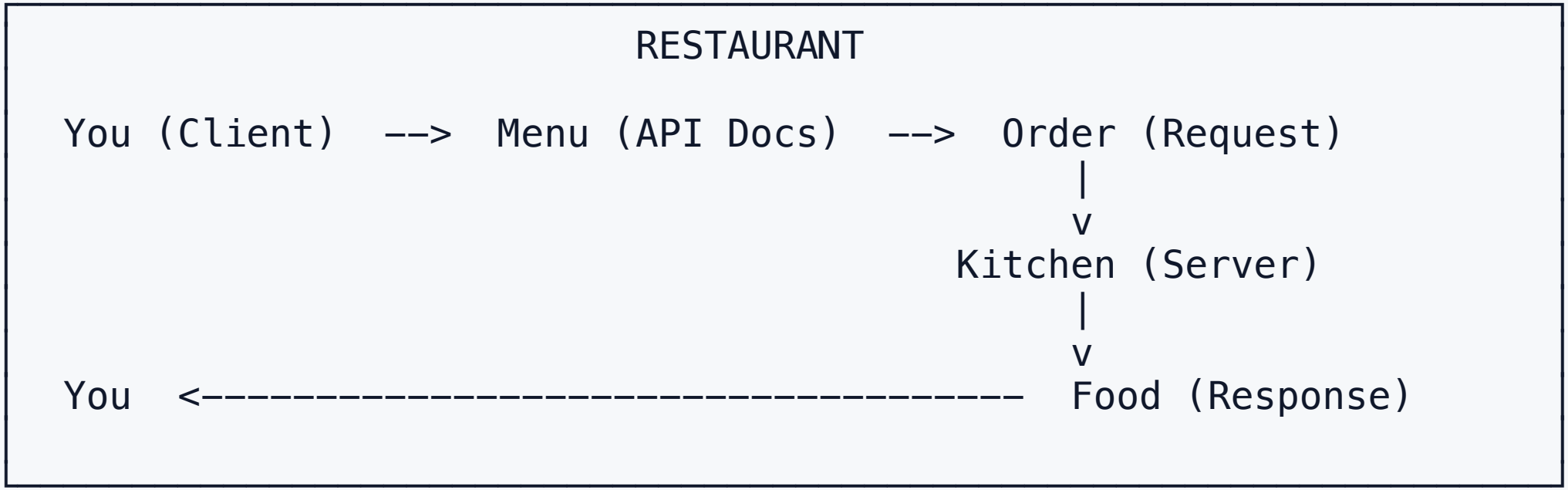
Data Needed	Source	Method
Movie titles, years	OMDb API	API calls
Ratings, genres	OMDb API	API calls
Budget, revenue	TMDb API	API calls
User reviews	IMDb website	Scraping
Critic reviews	Rotten Tomatoes	Scraping

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

Part 3: What is an API?

The contract between programs

API: A Restaurant Analogy



- **Menu** = API documentation (what you can order)
- **Order** = API request (what you're asking for)
- **Kitchen** = Server (processes your order)

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!)
SELECT * FROM movies WHERE title = 'Inception';

# With API (safe, controlled access)
GET /movies?title=Inception
```

APIs provide:

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

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).

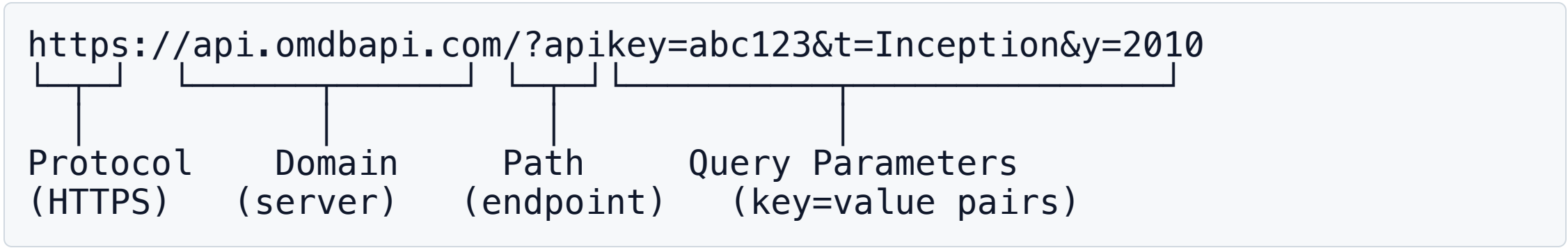
REST API: Key Principles

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:

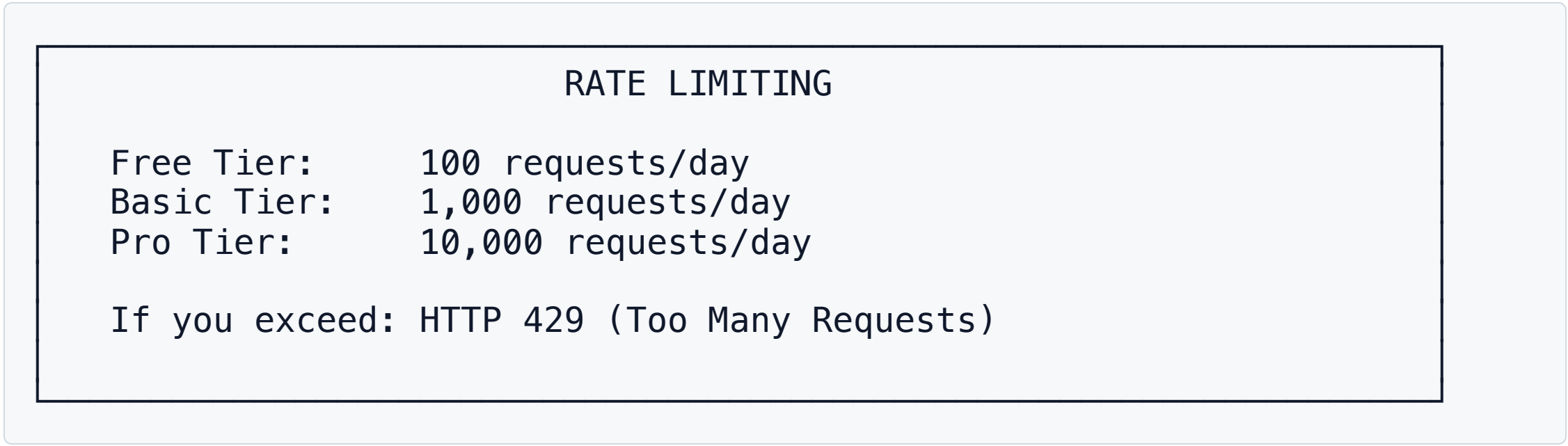
```
# 1. API Key in URL (simplest)
GET /movies?apikey=YOUR_KEY

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

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

Rate Limiting

Why? Servers have limited resources.



Rate limit headers in response:

- `X-RateLimit-Limit: 100`
- `X-RateLimit-Remaining: 42`

Part 4: HTTP Fundamentals

The language of the web

What is HTTP?

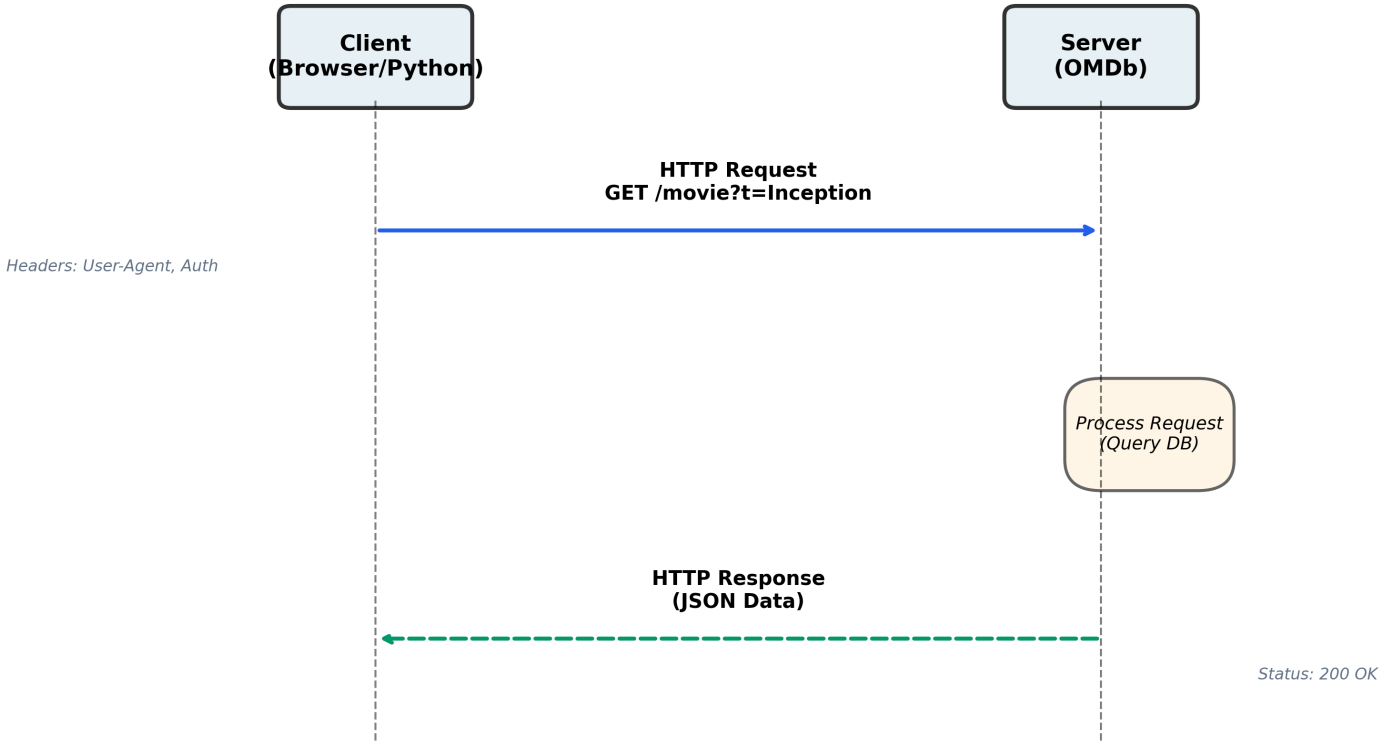
HTTP = **H**yper**T**ext **T**ransfer **P**rotocol

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)

The Client-Server Model



1. Client initiates connection
2. Client sends request

HTTP Request Structure

Every HTTP request has three parts:

```
1. REQUEST LINE
   GET /movies?t=Inception HTTP/1.1
```

```
2. HEADERS
   Host: api.omdbapi.com
   User-Agent: Python/3.9
   Accept: application/json
   Authorization: Bearer abc123
```

```
3. BODY (optional, for POST/PUT)
   {"title": "New Movie", "year": 2024}
```

HTTP Response Structure

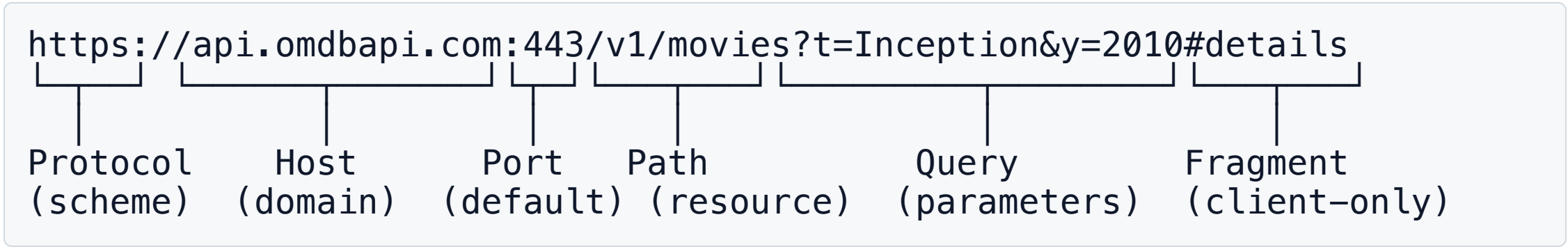
Every HTTP response has three parts:

```
1. STATUS LINE
   HTTP/1.1 200 OK

2. HEADERS
   Content-Type: application/json
   Content-Length: 1234
   X-RateLimit-Remaining: 99

3. BODY (the actual data)
   {"Title": "Inception", "Year": "2010", ...}
```

URL Anatomy



Components:

- **Protocol**: `http://` or `https://` (encrypted)
- **Host**: Domain name or IP address
- **Port**: Usually implicit (80 for HTTP, 443 for HTTPS)
- **Path**: Location of resource on server
- **Query**: Parameters as `key=value` pairs
- **Fragment**: Client-side anchor (not sent to server)

Common HTTP Headers

Request Headers (what client sends):

Header	Purpose	Example
`Host`	Target server	`api.omdbapi.com`
`User-Agent`	Client identification	`Mozilla/5.0`
`Accept`	Preferred response format	`application/json`
`Authorization`	Authentication	`Bearer token123`
`Content-Type`	Body format (POST)	`application/json`

Common HTTP Headers (Response)

Response Headers (what server sends):

Header	Purpose	Example
`Content-Type`	Body format	`application/json`
`Content-Length`	Size in bytes	`1234`
`Cache-Control`	Caching rules	`max-age=3600`
`X-RateLimit-Remaining`	API quota left	`99`
`Set-Cookie`	Session cookie	`session=abc123`

Part 5: HTTP Methods - GET and POST

The two most important verbs

HTTP Methods Overview

Method	Purpose	Has Body?	Safe?	Idempotent?
GET	Retrieve data	No	Yes	Yes
POST	Create/submit data	Yes	No	No
PUT	Replace resource	Yes	No	Yes
PATCH	Partial update	Yes	No	No
DELETE	Remove resource	No	No	Yes

For data collection: 90% GET, 10% POST

GET Request: Retrieving Data

Purpose: Fetch data without modifying anything.

```
GET /movies?t=Inception&y=2010 HTTP/1.1
Host: api.omdbapi.com
Accept: application/json
```

Characteristics:

- Parameters in URL (query string)
- No request body
- **Safe**: Doesn't change server state
- **Idempotent**: Same request = same result
- **Cacheable**: Responses can be cached

POST Request: Sending Data

Purpose: Submit data to create or process something.

```
POST /api/feedback HTTP/1.1
Host: example.com
Content-Type: application/json

{"movie_id": 123, "rating": 5, "review": "Great!"}
```

Characteristics:

- Data in request body (not URL)
- **Not safe**: Modifies server state
- **Not idempotent**: Multiple POSTs create multiple resources
- **Not cacheable**

GET vs POST: When to Use Which

Scenario	Method	Why
Fetching movie details	GET	Retrieving data
Searching for movies	GET	Query in URL
Submitting a review	POST	Creating new data
Uploading an image	POST	Sending binary data
User login	POST	Sensitive data in body
Listing all movies	GET	No modification

Data Collection = Mostly GET

Data Submission = POST

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

Success (2xx):

- `200 OK` - Request succeeded
- `201 Created` - Resource created (POST)
- `204 No Content` - Success, empty body

Client Errors (4xx):

- `400 Bad Request` - Malformed request
- `401 Unauthorized` - Missing authentication
- `403 Forbidden` - Not allowed
- `404 Not Found` - Resource doesn't exist
- `429 Too Many Requests` - Rate limited

Part 6: Response Formats

Same data, different representations

Why Different Formats?

```
Same movie data can be represented in:

- 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

Format 1: JSON

The most common API format today.

```
{
  "title": "Inception",
  "year": 2010,
  "genres": ["Sci-Fi", "Action", "Thriller"],
  "director": {
    "name": "Christopher Nolan",
    "nationality": "British"
  },
  "rating": 8.8,
  "in_production": false
}
```

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"`

Format 2: XML

The enterprise standard (still used in SOAP APIs).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<movie>
  <title>Inception</title>
  <year>2010</year>
  <genres>
    <genre>Sci-Fi</genre>
    <genre>Action</genre>
  </genres>
  <director nationality="British">
    Christopher Nolan
  </director>
  <rating>8.8</rating>
</movie>
```

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.

```
title,year,genre,director,rating
Inception,2010,Sci-Fi,Christopher Nolan,8.8
Avatar,2009,Action,James Cameron,7.9
The Matrix,1999,Sci-Fi,Wachowskis,8.7
```

Pros:

- Opens in Excel/Google Sheets
- Easy to load into pandas: `pd.read_csv("movies.csv")`
- Very compact

Cons:

- Flat structure only (no nesting)

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;
}
```

Binary on the wire (not human-readable):

```
0a 09 49 6e 63 65 70 74 69 6f 6e 10 da 0f ...
```

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: Chrome DevTools

Your window into HTTP traffic

Why Chrome DevTools?

DevTools lets you see:

- Every HTTP request your browser makes
- Request headers, body, timing
- Response headers, body, status codes
- Copy requests as curl commands!

This is how you learn what APIs a website uses.

Opening DevTools

Three ways to open:

1. **Keyboard**: `F12` or `Ctrl+Shift+I` (Windows/Linux) / `Cmd+Option+I` (Mac)
2. **Right-click**: Right-click on page → "Inspect"
3. **Menu**: Chrome menu → More Tools → Developer Tools

Navigate to the "Network" tab

The Network Tab

```
Network
[*] Preserve log   [ ] Disable cache   [Filter]
Name           Status   Type      Size     Time
api/movies     200      fetch     1.2 KB   45 ms
styles.css     200      css       5.4 KB   23 ms
logo.png       200      image     15 KB    67 ms
analytics.js   200      script    8.1 KB   89 ms
```

Every row = one HTTP request/response

Filtering Requests

Filter by type:

Filter	Shows
All	Everything
Fetch/XHR	API calls (AJAX) ← Most useful!
Doc	HTML documents
CSS	Stylesheets
JS	JavaScript files
Img	Images

Click "Fetch/XHR" to see only API calls

Inspecting a Request

Click on any request to see details:

```
Headers   Preview   Response   Timing   Cookies
```

```
General:
  Request URL: https://api.example.com/movies?id=123
  Request Method: GET
  Status Code: 200 OK
```

```
Response Headers:
  content-type: application/json
  x-ratelimit-remaining: 99
```

```
Request Headers:
  authorization: Bearer eyJhbGc...
  user-agent: Mozilla/5.0...
```

The Preview & Response Tabs

Preview Tab: Formatted JSON viewer

```
{
  "title": "Inception",
  "year": 2010,
  "rating": 8.8
}
```

Response Tab: Raw response body

```
{"title":"Inception","year":2010,"rating":8.8}
```

Copy as curl

The most powerful feature!

1. Right-click on any request
2. Select "Copy" → "Copy as cURL"
3. Paste into terminal

```
curl 'https://api.example.com/movies?id=123' \
  -H 'accept: application/json' \
  -H 'authorization: Bearer eyJhbGciOiJIUzI1...' \
  -H 'user-agent: Mozilla/5.0 (Macintosh...)' \
  --compressed
```

Now you can replay the exact request from your terminal!

Demo: Finding Hidden APIs

Many websites use hidden APIs. Here's how to find them:

1. Open DevTools → Network tab
2. Filter by "Fetch/XHR"
3. Interact with the website (search, click, load more)
4. Watch for API calls appearing in the list
5. Click on interesting requests to inspect them
6. Copy as curl to test in terminal

Example: Search on IMDb and watch for API calls...

DevTools Pro Tips

Preserve log: Keep requests when navigating between pages

Disable cache: See fresh requests every time

Search: `Ctrl+F` to search in all requests

Filter by URL: Type in filter box to match URLs

Clear: Click the clear icon to clear all requests

Throttling: Simulate slow networks (3G, offline)

Part 8: 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://api.omdbapi.com/?apikey=demo&t=Inception"
```

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`

curl: GET Request

```
# Simple GET request
curl "https://api.omdbapi.com/?apikey=demo&t=Inception"
```

Output:

```
{"Title":"Inception","Year":"2010","Rated":"PG-13",
"Released":"16 Jul 2010","Runtime":"148 min",...}
```

Important: Quote the URL! (prevents shell interpretation of `&`)

curl: Adding Headers

```
curl "https://api.example.com/movies" \
     -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://api.omdbapi.com/?apikey=demo&t=Inception"
```

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://api.omdbapi.com/?apikey=demo&t=Inception"
```

Shows everything (request AND response):

```
> GET /?apikey=demo&t=Inception HTTP/2
> Host: api.omdbapi.com
> User-Agent: curl/7.79.1
> Accept: */*
>
< HTTP/2 200
< content-type: application/json
< 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://api.omdbapi.com/?apikey=demo&t=Inception" | jq .
```

Output (formatted):

```
{
  "Title": "Inception",
  "Year": "2010",
  "Rated": "PG-13",
  "Runtime": "148 min",
  "Genre": "Action, Adventure, Sci-Fi"
}
```

jq: Extracting Specific Fields

```
# Get just the title
curl -s ... | jq '.Title'
# Output: "Inception"

# Get multiple fields as new object
curl -s ... | jq '{title: .Title, year: .Year, rating: .imdbRating}'
# Output: {"title": "Inception", "year": "2010", "rating": "8.8"}

# Get first element of array
curl -s ... | jq '.Search[0]'

# Get all titles from array
curl -s ... | jq '.Search[].Title'
```

curl: Saving to File

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

# Silent mode (no progress bar)
curl -s "https://api.example.com/data" -o output.json

# Save with pretty formatting
curl -s ... | jq . > formatted.json
```

curl: POST Request

```
curl -X POST "https://api.example.com/reviews" \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer YOUR_TOKEN" \
     -d '{"movie_id": 123, "rating": 5, "review": "Amazing!"}'
```

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

```
# Form-encoded data (like HTML forms)
curl -X POST "https://example.com/login" \
     -d "username=john" \
     -d "password=secret"

# Equivalent to:
curl -X POST "https://example.com/login" \
     -H "Content-Type: application/x-www-form-urlencoded" \
     -d "username=john&password=secret"
```

curl: File Upload

```
# Upload a file
curl -X POST "https://api.example.com/upload" \
     -F "file=@/path/to/image.jpg" \
     -F "description=My photo"
```

`-F` = **multipart form data** (for file uploads)

`@` = **read from file**

curl: Useful Options

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

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

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

# Fail silently on HTTP errors
curl -f "https://api.example.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

```python
import requests

# Make the request
response = requests.get(
    "https://api.omdbapi.com/",
    params={"apikey": "demo", "t": "Inception"}
)

# Check status
print(response.status_code)  # 200

# Get JSON data
data = response.json()
print(data["Title"])  # "Inception"
print(data["Year"])   # "2010"
```

requests: Using params

Don't manually build query strings!

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

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

Python handles URL encoding automatically!

requests: Adding Headers

```
response = requests.get(
    "https://api.example.com/movies",
    headers={
        "Authorization": "Bearer YOUR_TOKEN",
        "Accept": "application/json",
        "User-Agent": "MyApp/1.0"
    }
)
```

requests: Response Object

```
response = requests.get("https://api.omdbapi.com/...")

# Status code
response.status_code        # 200

# Headers (dict-like)
response.headers["Content-Type"]  # "application/json"

# Body as text
response.text               # '{"Title": "Inception"...}'

# Body as JSON (parsed dict)
response.json()             # {"Title": "Inception", ...}

# Was it successful?
response.ok                 # True (for 2xx status codes)
```

requests: POST with JSON

```
import requests

response = requests.post(
    "https://api.example.com/reviews",
    headers={
        "Authorization": "Bearer YOUR_TOKEN"
    },
    json={  # Use json= for automatic JSON encoding
        "movie_id": 123,
        "rating": 5,
        "review": "Great movie!"
    }
)

if response.status_code == 201:
    print("Review submitted!")
    print(response.json())
```

requests: POST with Form Data

```
# Form-encoded POST (like HTML forms)
response = requests.post(
    "https://example.com/login",
    data={  # Use data= for form encoding
        "username": "john",
        "password": "secret"
    }
)
```

Remember:

- `json=` → Content-Type: application/json
- `data=` → Content-Type: application/x-www-form-urlencoded

requests: Error Handling

```python
import requests

try:
    response = requests.get(
        "https://api.omdbapi.com/",
        params={"apikey": "demo", "t": "Inception"},
        timeout=10  # seconds
    )

    # Raise exception for 4xx/5xx status codes
    response.raise_for_status()

    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}")
```

requests: Looping Over Multiple Items

```
import requests
import time

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

for title in movies:
    response = requests.get(
        "https://api.omdbapi.com/",
        params={"apikey": "YOUR_KEY", "t": title}
    )

    if response.ok:
        results.append(response.json())
        print(f"Got: {title}")

    time.sleep(0.5)  # Be nice to the server!

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

requests: Session for Multiple Requests

```python
import requests

# Session persists settings across requests
session = requests.Session()
session.headers.update({
    "Authorization": "Bearer YOUR_TOKEN",
    "User-Agent": "MyApp/1.0"
})

# Now all requests use these headers
r1 = session.get("https://api.example.com/movies")
r2 = session.get("https://api.example.com/reviews")
r3 = session.get("https://api.example.com/users")

# Also maintains cookies automatically!
```

requests: Practical Example

```
import requests
import pandas as pd

def fetch_movie_data(titles, api_key):
    """Fetch movie data for a list of titles."""
    movies = []

    for title in titles:
        response = requests.get(
            "https://api.omdbapi.com/",
            params={"apikey": api_key, "t": title},
            timeout=10
        )

        if response.ok and response.json().get("Response") == "True":
            movies.append(response.json())

    return pd.DataFrame(movies)

# Usage
df = fetch_movie_data(["Inception", "Avatar"], "YOUR_KEY")
print(df[["Title", "Year", "imdbRating"]])
```

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

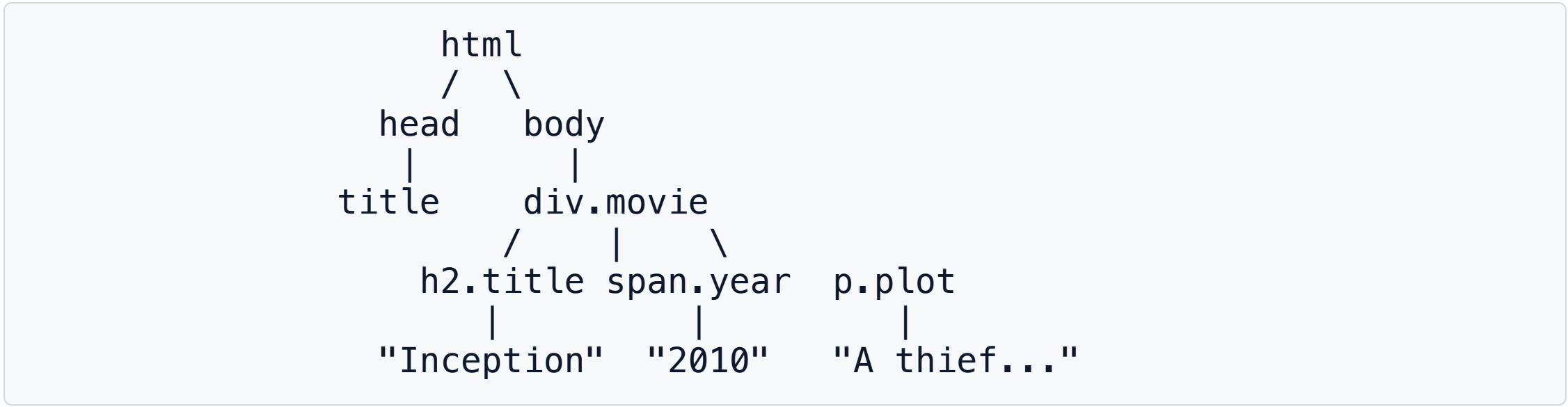
Rule: Always prefer APIs when available.

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
```

```
from bs4 import BeautifulSoup
import requests

# Fetch the page
response = requests.get("https://example.com/movies")
html = response.text

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

# Now we can search!
```

BeautifulSoup: Finding Elements

```
from bs4 import BeautifulSoup

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"

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

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)

# Complex selectors
links = soup.select('a[href^="/movies/"]')  # href starts with
```

BeautifulSoup: Extracting Data

```
# Get text content
element.text            # "Inception"
element.get_text()      # Same, with options
element.get_text(strip=True)  # Remove whitespace

# Get attribute
link = soup.select_one('a')
link.get('href')        # "/movies/123"
link['href']            # Same thing

# Get all attributes
link.attrs              # {'href': '/movies/123', 'class': ['btn']}
```

Scraping Example: Movie List

```
import requests
from bs4 import BeautifulSoup

url = "https://example.com/top-movies"
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(),
        'rating': card.select_one('.rating').text.strip(),
    }
    movies.append(movie)

print(f"Found {len(movies)} movies")
```

Handling Pagination

```
import requests
from bs4 import BeautifulSoup
import time

base_url = "https://example.com/movies?page="
all_movies = []

for page in range(1, 11):  # Pages 1-10
    response = requests.get(f"{base_url}{page}")
    soup = BeautifulSoup(response.text, 'html.parser')

    movies = soup.select('.movie-card')
    if not movies:
        break  # No more pages

    for m in movies:
        all_movies.append(m.select_one('.title').text)

    print(f"Page {page}: {len(movies)} movies")
    time.sleep(1)  # Be polite!
```

Scraping Ethics & Best Practices

```
import time
import requests

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

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

Rules:

1. Check `robots.txt` first
2. Add delays between requests
3. Identify yourself (User-Agent)

Checking robots.txt

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

```
User-agent: *
Disallow: /search/
Disallow: /ap/
Allow: /title/
Crawl-delay: 1
```

Meaning:

- Cannot scrape `/search/` pages
- Can scrape `/title/` pages (movie pages)
- Wait 1 second between requests

Part 11: Putting It All Together

Back to our Netflix mission

Remember Our Goal?

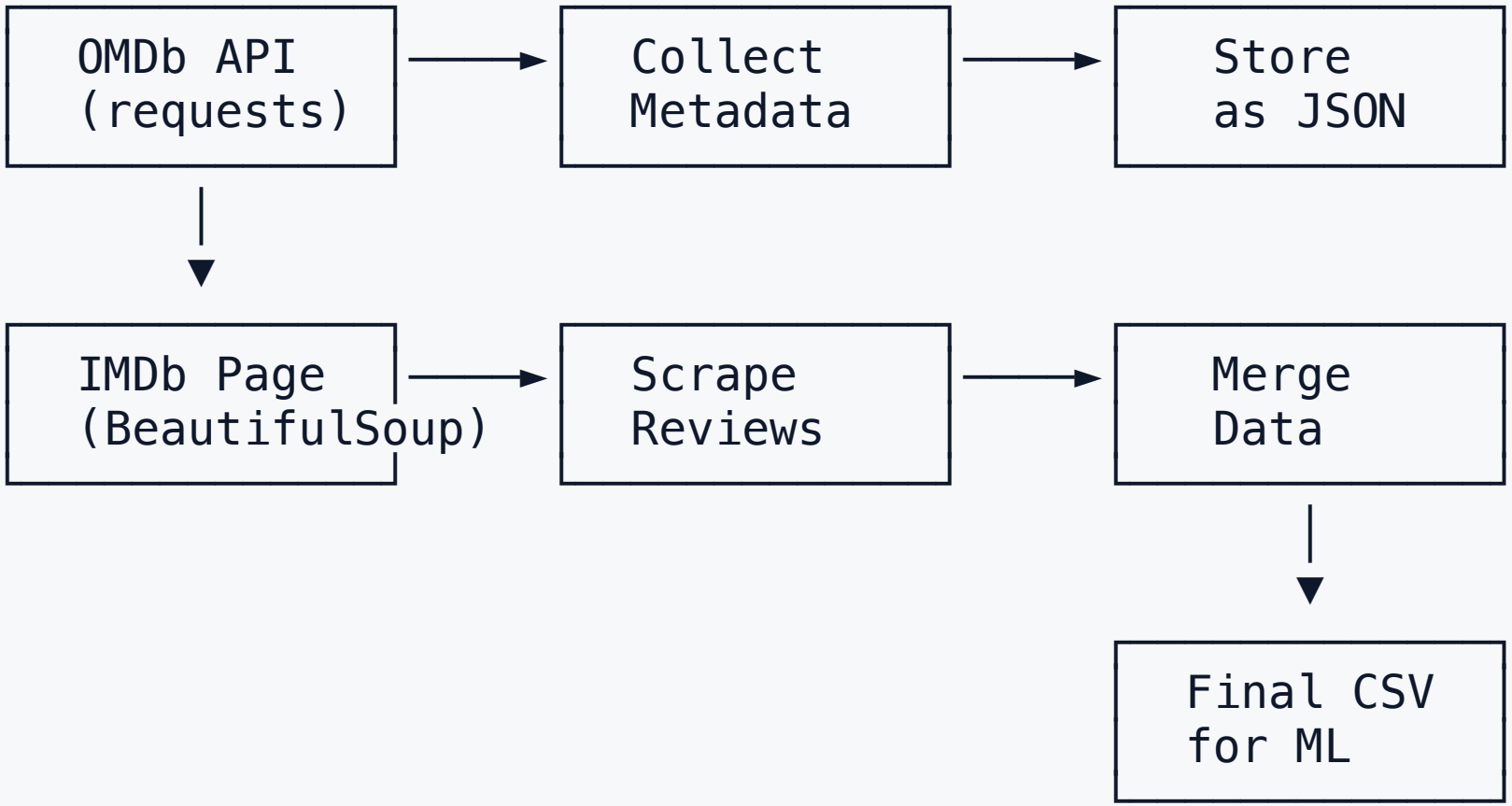
Build a dataset for movie success prediction:

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

```
import requests
import time

API_KEY = "your_omdb_key"
movies_to_fetch = ["Inception", "Avatar", "The Matrix"]
results = []

for title in movies_to_fetch:
    response = requests.get(
        "https://api.omdbapi.com/",
        params={"apikey": API_KEY, "t": title}
    )

    if response.ok:
        data = response.json()
        if data.get("Response") == "True":
            results.append(data)

    time.sleep(0.5)  # Rate limiting

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

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: https://curl.se/docs/
- requests: https://requests.readthedocs.io/
- BeautifulSoup: https://beautiful-soup-4.readthedocs.io/

Free APIs for Practice:

- JSONPlaceholder: https://jsonplaceholder.typicode.com/
- OMDb: https://www.omdbapi.com/
- Public APIs list: https://github.com/public-apis/public-apis

Questions?

Thank You!

See you in the lab!