Understanding how to collect data from **APIs (Application Programming Interfaces)** is essential for data science, as they provide real-time, structured access to web data.

## 1. Understanding REST APIs and Endpoints 🌐

Theory:

A REST (Representational State Transfer) API is a set of rules that governs how two systems (your Python script and a web service) communicate over the internet.

| Concept | Description | Example |
| --- | --- | --- |
| **Resource** | Any object or entity the API provides access to (e.g., a customer, a product, a tweet). | /users, /products, /orders |
| **Endpoint** | A specific URL where an API retrieves a resource. It is the target of a request. | https://api.example.com/v1/products/42 |
| **HTTP Methods** | Standard actions (verbs) used to interact with a resource at an endpoint. | **GET** (Retrieve data), **POST** (Create data), **PUT/PATCH** (Update data), **DELETE** (Remove data). **Data Collection always uses GET.** |
| **JSON** | **(JavaScript Object Notation)** The standard, language-agnostic format for sending and receiving data between a client and a REST API. It structures data as key-value pairs. | {"id": 42, "name": "Laptop", "price": 1200.00} |

## 2. Using Python for API Calls with requests and json 🐍

The **requests** library is the standard Python tool for making HTTP requests. The **json** library handles the decoding of the API's response.

### Python Example: GET Request

Python

import requests  
import json # Used implicitly by requests, but good to know  
  
API\_ENDPOINT = 'https://api.example.com/v1/products'  
# Parameters to filter the request (e.g., search for items on sale)  
params = {'status': 'onsale', 'limit': 100}   
  
try:  
 # Make the GET request  
 response = requests.get(API\_ENDPOINT, params=params)  
   
 # Raise an exception for bad status codes (4xx or 5xx)  
 response.raise\_for\_status()   
  
 # Decode the JSON response into a Python dictionary/list  
 data = response.json()  
   
 # Process the data (e.g., load into a pandas DataFrame)  
 print(f"Successfully retrieved {len(data)} records.")  
 # df = pd.DataFrame(data)   
   
except requests.exceptions.RequestException as e:  
 print(f"API Request Failed: {e}")

## 3. Handling Authentication Mechanisms 🔑

APIs typically require authentication to verify your identity and control access to data.

| Mechanism | Description | Python Implementation |
| --- | --- | --- |
| **API Keys** | A simple secret string (token) passed with the request. | Often passed as a **query parameter** or in the **HTTP Header**. requests.get(url, params={'api\_key': 'your\_key'}) or requests.get(url, headers={'Authorization': 'Bearer your\_key'}) |
| **OAuth** | A complex token-based system used by major platforms (Google, Twitter). It involves obtaining a short-lived **access token** after a user grants permission. | Tokens are typically passed in the **Authorization Header** as a Bearer token: headers={'Authorization': 'Bearer ' + access\_token}. |

## 4. Pagination and Rate-Limiting for Large Datasets 🛑

These are mandatory concepts when collecting large volumes of data from an API.

### A. Pagination

Theory:

APIs rarely return millions of records in a single request. Instead, they use pagination to split large results into smaller, manageable "pages" or "batches."

| Pagination Type | Description |
| --- | --- |
| **Offset/Limit** | Uses an offset (starting point) and a limit (items per page). You loop by increasing the offset until no more data is returned. |
| **Cursor/Token** | The API returns a next\_page\_token (or cursor) with the current page's data. You must pass this token to request the next page. **More efficient for large datasets.** |

Python Implementation Strategy:

You use a while loop to continuously make requests, updating the offset or token in the params dictionary for each iteration until the API returns an empty list or no next\_page\_token.

### B. Rate-Limiting

Theory:

APIs enforce rate limits to control the number of requests a user can make within a specified time frame (e.g., 100 requests per minute). This prevents server overload.

* **Handling:** When you hit the limit, the API returns a **429 (Too Many Requests)** status code.
* **Best Practice:** Check the response **HTTP Headers** (e.g., X-RateLimit-Remaining, Retry-After) to know how long to wait. Use Python's time.sleep() function to pause your script for the specified duration before making the next request. This is often called a **"polite" scraping** strategy.