Clase 047 — APIs REST con requests

Parte: 0 — Prerrequisitos · Fuente: HTTP: The Definitive Guide caps. 1-2 · requests docs. · Duración estimada: 90 min.

Clase 047 — APIs REST con requests

> *Parte: 0 — Prerrequisitos · Fuente: HTTP: The Definitive Guide caps. 1-2 · requests docs. · Duración estimada: 90 min.*

Objetivo

Que el alumno consuma APIs REST públicas con requests: GET con parámetros, manejo de status codes, autenticación (header, bearer token), paginación, rate limiting con Retry, y carga eficiente con Session. Lo mínimo para no romper la API del proveedor ni tu pipeline.

Resultados de aprendizaje

Al finalizar la clase, el alumno podrá:

1. Hacer GET/POST con requests, manejar params, headers, body JSON.
2. Verificar status code (200 vs 4xx vs 5xx) y usar raise_for_status().
3. Autenticarse con header Authorization: Bearer ... o API key en header/query.
4. Paginar correctamente cuando la API devuelve resultados en páginas.
5. Rate-limiting con urllib3.util.retry.Retry para reintentos exponenciales.
6. Reusar conexión con requests.Session para múltiples requests.

Temas

#	Tema	Por qué importa
1	Métodos HTTP: GET, POST, PUT, DELETE	Verbos REST.
2	Status codes: 2xx/3xx/4xx/5xx	Cómo reaccionar a cada uno.
3	Params, headers, body	Las 3 formas de mandar datos.
4	Autenticación: Bearer token, API key	Header Authorization.
5	Paginación: offset/limit, cursor, link hea	3 patrones comunes.
6	Rate limiting + retry exponencial	No tirar la API ajena.
7	requests.Session para reuso	Más rápido + cookies persistentes.

Versión profundizada — 2026

El tema moderno que vivía como complemento dentro de esta clase ahora tiene clase propia dedicada:

- Clase 046a — async / httpx / aiohttp para data scientists

Definiciones y características

REST (REpresentational State Transfer)

: Estilo arquitectónico para APIs web sobre HTTP. Recursos identificados por URLs, operaciones por verbos HTTP (GET=leer, POST=crear, PUT=update, DELETE=borrar).

requests

: Librería Python de facto para hacer HTTP. API simple: requests.get(url, params=..., headers=..., timeout=...). Soporta auth, cookies, sessions, retry.

Status code

: Número HTTP que indica resultado: 2xx éxito, 3xx redirect, 4xx error cliente (404 no encontrado, 401 no auth, 403 prohibido, 429 rate limited), 5xx error servidor.

Paginación

: API que devuelve resultados en bloques (no todo de golpe). Patrones: offset/limit (?page=2), cursor (?after=<id>), Link header (<url>; rel="next").

Rate limiting

: Política del servidor: máximo N requests/seg/usuario. Excederlo → 429. Respétalo con delays y retries exponenciales.

Session

: Reuso de conexión TCP/TLS entre requests. Mantiene cookies. ~10× más rápido para múltiples requests al mismo host vs requests.get repetido.

Bearer token

: Esquema de auth común: Authorization: Bearer <token> header. Token suele ser JWT o opaque string emitido por OAuth/login.

Dataset / recursos

API pública sin auth: https://api.coingecko.com (precios cripto). Sin API key necesaria.

Ejercicios

1. GET básico. requests.get('https://api.github.com'). Inspecciona status_code, headers, .json().

2. Con params. GitHub search: https://api.github.com/search/repositories?q=python+ml&sort=stars. Imprime top 5.

3. raise_for_status + try. Pega a una URL que devuelve 404 (/notfound) y maneja la excepción.

4. Paginación. GitHub events API. Itera 3 páginas con page=1,2,3.

5. Session + Retry. Configura una Session con HTTPAdapter + Retry (3 intentos, backoff 1s). Verifica que reintenta en 5xx simulado.

Homework verificable

Notebook que: (a) consulta una API pública (CoinGecko, GitHub, JSONPlaceholder) con GET; (b) maneja status codes con try/except; (c) pagina 3+ páginas; (d) configura Session con Retry exponencial; (e) reporta cuánto se tardó vs un loop sin Session.

Criterio de aceptación: Maneja al menos un error sin crash. Pagination devuelve datos esperados.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
requests.exceptions.ConnectTimeout o Timeo	API lenta o sin red. Fix: siempre pasa tim
API responde 200 pero .json() lanza error	Body no es JSON (HTML de error, vacío). Fi
Hardcodeé el token en el código y subí a G	Catástrofe de seguridad — el token es públ
Mi script tira la API ajena (HTTP 429)	Sin rate limiting. Fix: time.sleep() entre
HTTPError no se lanza con status 4xx	requests NO lanza por default. Fix: r.rais
RuntimeError: asyncio.run() cannot be call	El notebook ya tiene un event loop corrien

Preguntas frecuentes

¿requests o httpx?

requests sigue siendo la default (estable, ubícua). httpx drop-in con async support (async with httpx.AsyncClient() as client:). Para async/HTTP2, httpx; para todo lo demás, requests.

¿Cuándo Session?

Más de 2-3 requests al mismo host. La primera request hace handshake TCP/TLS (~100ms); Session lo reusa. Para single request, no aporta.

¿json= o data= en POST?

json=dict: serializa a JSON y setea Content-Type: application/json. data=dict: form-encoded (application/x-www-form-urlencoded). Para APIs REST modernas, casi siempre json=.

¿Cómo paginar genéricamente?

Loop hasta que la API diga "no más": while True: r = requests.get(url, params=...); items.extend(r.json()['data']); if not r.json().get('next'): break.

¿Auth OAuth desde Python?

Para casos simples (Bearer fijo): pasa el header. Para OAuth flow completo: authlib, requests-oauthlib. Para producción: librería oficial del proveedor (google-auth, pyOpenSSL, etc.).

¿Cuándo paso de requests a httpx async?

Regla práctica: cuando tenés >20 requests simultáneos al mismo proveedor y el cuello de botella es latencia de red (no CPU). Si tu script está 90% del tiempo esperando respuestas HTTP, async te da un speedup de 10-100×. Si en cambio estás parseando JSONs gigantes o haciendo cálculo pesado, async no ayuda — ahí lo tuyo es multiprocessing. Pocas requests (<10): seguí con requests, no vale la complejidad.

Referencias

- requests docs
- urllib3 Retry
- GitHub REST API
- HTTP status codes (MDN)
- httpx docs
- asyncio + Jupyter (autoawait)

Siguiente clase

Clase 048 — Web scraping con BeautifulSoup

Apéndice: notebook (primer bloque)

Primera celda ejecutable del notebook de la clase.

```
import requests
import time
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry
print(f'requests: {requests.__version__}')
```

Archivos complementarios

- notebook.ipynb