

Pagination, erreurs, 401 et 429

Un script qui marche sur la première page, avec un serveur de bonne humeur, c'est facile. Un script robuste doit aussi gérer la pagination, les erreurs et un serveur qui demande de ralentir.

La pagination

Une API ne renvoie jamais une liste de 10 000 éléments d'un coup : elle découpe en **pages**. Sur API-ritif, une liste ressemble à ceci :

```
{
  "items": [ {"id": 1, "name": "Hello World"}, ... ],
  "page": 1,
  "per_page": 5,
  "total": 13,
  "_links": {
    "self": {"href": "/v1/cocktails?page=1&per_page=5"},
    "next": {"href": "/v1/cocktails?page=2&per_page=5"},
    "last": {"href": "/v1/cocktails?page=3&per_page=5"}
  }
}
```

La méthode la plus fiable : **suivre le lien** `next` jusqu'à ce qu'il n'y en ait plus. Pas besoin de calculer les numéros de page.

```
import pprint
from urllib.parse import urljoin

def get_all(path, headers=None):
    items = []
    url = urljoin(BASE, path)
    while url:
        res = requests.get(url, headers=headers)
```

```
res.raise_for_status()
data = res.json()
items.extend(data["items"])
next_link = data["_links"].get("next")
url = urljoin(BASE, next_link["href"]) if next_link else None
return items

cocktails = get_all("/v1/cocktails")

pprint.pp(cocktails)
```

Les liens sont relatifs (`/v1/...`) : `urljoin()` les combine avec l'adresse du serveur.

Lire les erreurs

API-ritif renvoie ses erreurs au format `application/problem+json` (détaillé au chapitre 5). Le champ `detail` est fait pour être affiché :

```
{
  "type": "https://api-ritif.example/problems/out-of-stock",
  "title": "Rupture de stock",
  "status": 409,
  "detail": "Le 404 Spritz est introuvable. Comme d'habitude."
}
```

```
if not res.ok:
    problem = res.json()
    print(problem["status"], problem["title"], ":", problem["detail"])
```

401 : l'authentification a échoué

Un `401` signifie que le serveur ne vous reconnaît pas : clé absente, clé fausse, ou **token expiré**. Certains tokens ne vivent que quelques minutes (c'est le cas des JWT d'API-ritif, voir chapitre 8).

Réaction attendue d'un client :

- clé absente ou fausse : inutile d'insister, il faut corriger la configuration ;

- token expiré : en redemander un nouveau, puis **rejouer** la requête une fois.

429 : le serveur demande de ralentir

Pour se protéger, une API limite le nombre de requêtes par client (*rate limiting*). Au-delà, elle répond `429 Too Many Requests`, souvent avec un header `Retry-After` qui indique combien de secondes attendre.

```
HTTP/1.1 429 Too Many Requests
Retry-After: 10
Content-Type: application/problem+json

{"title": "Too Many Requests", "status": 429, "detail": "Le barman est débordé, reviens dans 10 s"}
```

Un bon client **attend**, puis réessaie :

```
import time

def polite_get(url, **kwargs):
    while True:
        res = requests.get(url, **kwargs)
        if res.status_code != 429:
            return res
        wait = int(res.headers.get("Retry-After", 1))
        print("429 reçu, pause de", wait, "s")
        time.sleep(wait)
```

Sans `Retry-After`, la bonne pratique est le **backoff exponentiel** : attendre 1 s, puis 2 s, puis 4 s, puis 8 s... Le serveur respire, et vous finissez par passer.

Réessayer immédiatement en boucle après un `429`, c'est comme klaxonner dans un embouteillage : ça n'avance pas plus vite, et tout le monde vous déteste.