httpx

An overview of httpx, the modern client that does sync and async in one API, with HTTP/2, streaming, and connection reuse.

Sync Requests

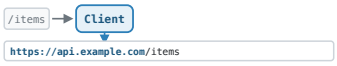
Top-level functions for one-off calls; identical shape to requests.

PURPOSE	COMMAND	RESULT
Read a resource.	`httpx.get("https://api.example.com/items")`	
Create with a JSON body.	`httpx.post(url, json={"name": "ada"})`	
Add query params.	`httpx.get(url, params={"q": "test", "page": 2})`	
Set custom headers.	`httpx.get(url, headers={"User-Agent": "my-app/1.0"})`	
Other verbs.	`httpx.put(url, json=d), httpx.patch(url, json=d),httpx.delete(url)`	
Read the response.	`r.status_code,r.json(),r.text`	

`httpx.get / .post / .put / .patch / .delete mirror the requests API; reach for these for one-off calls, a Client for anything repeated`

The reusable Client

One Client pools connections, sets base_url and default headers.

PURPOSE	COMMAND	RESULT
Open a client (context block).	`with httpx.Client() as client: ...`	
Set a base URL once.	`httpx.Client( base_url="https://api.example.com")`	
Default headers for every call.	`httpx.Client(headers={ "Authorization": "Bearer TOKEN"})`	
Make requests through it.	`client.get("/items") client.post("/items", json=d)`	
Size the connection pool.	`httpx.Client(limits=httpx.Limits(max_connections=100, max_keepalive_connections=20))`	
Cookies persist across calls.	`client.get(login_url);client.get(profile_url)`	

`one httpx.Client() pools TCP connections and carries base_url, default headers, cookies, and pool limits across every call`

Bodies & auth

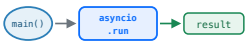
json= / data= / files= / content=, plus built-in auth flows.

PURPOSE	COMMAND	RESULT
Send JSON (auto-serialize).	`httpx.post(url, json={"id": 1, "name": "ada"})`	
Send a form (urlencoded).	`httpx.post(url, data={"key": "value"})`	
Upload a file (multipart).	`httpx.post(url, files={"file": open("a.png", "rb")})`	
Send raw bytes/text.	`httpx.post(url, content=b"raw bytes")`	
HTTP Basic auth.	`httpx.get(url, auth=("user", "passwd"))`	
HTTP Digest auth.	`httpx.get(url, auth=httpx.DigestAuth("u", "p"))`	

json= sets application/json, data= sends a form, files= sends multipart; raw bytes/text go in content=, not data=

Async with AsyncClient + await

The same API, awaited; this is what httpx adds over requests.

PURPOSE	COMMAND	RESULT
Open an async client.	`async with httpx.AsyncClient() as client:`	
Await a single request.	`r = await client.get(url)`	
Same verbs, all awaited.	`await client.post(url, json=d)` `await client.delete(url)`	
Run from sync code.	`asyncio.run(main())`	
Read the response (no await).	`r.json(), r.text, r.status_code`	
Stream the body asynchronously.	`async with client.stream("GET", url) as r:` `async for chunk in r.aiter_bytes():`	

AsyncClient mirrors Client: every request is a coroutine you await, run from sync code with asyncio.run; reading the Response needs no await

Concurrent Fan-out

Launch many requests at once and gather them; this is the payoff.

PURPOSE	COMMAND	RESULT
Build the request coroutines.	`tasks = [client.get(u) for u in urls]`	
Run them all concurrently.	`responses = await asyncio.gather(*tasks)`	
Sequential vs concurrent timing.	`5x /delay/1: serial ~5s` `vs gather ~1.3s`	
Cap concurrency politely.	`sem = asyncio.Semaphore(10)` `async with sem: await client.get(u)`	
Stream results as they finish.	`for coro in asyncio.as_completed(tasks):` `r = await coro`	
Keep failures from sinking the batch.	`await asyncio.gather(*tasks,` `return_exceptions=True)`	

asyncio.gather(*tasks) over one pooled AsyncClient runs requests concurrently; Semaphore caps pressure, return_exceptions=True keeps one failure from sinking the batch

HTTP/2

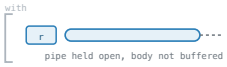
One flag multiplexes many streams over a single connection.

PURPOSE	COMMAND	RESULT
Install the HTTP/2 extra.	`pip install "httpx[http2]"`	
Turn on HTTP/2.	`client = httpx.Client(http2=True)`	
Confirm the negotiated version.	`r.http_version`	
Multiplex over one connection.	`httpx.AsyncClient(http2=True)`	
Falls back gracefully.	`r.http_version == "HTTP/1.1"`	
Verify status and version.	`print(r.status_code, r.http_version)`	

install httpx[http2], pass http2=True, then check r.http_version; httpx multiplexes over one connection and falls back to HTTP/1.1 transparently

Streaming responses

Don't load huge bodies into memory; iterate in chunks.

PURPOSE	COMMAND	RESULT
Open a streaming response.	`with client.stream("GET", url) as r:`	
Iterate raw bytes (downloads).	`for chunk in r.iter_bytes(chunk_size=8192):`	
Iterate decoded text lines.	`for line in r.iter_lines():`	
Read the status before the body.	`r = client.send(req, stream=True)`	
Save a stream to disk.	`with open("out.bin","wb") as f:` `f.writelines(r.iter_bytes())`	
Async streaming twin.	`async for chunk in r.aiter_bytes():`	

client.stream("GET", url) inside a with block; pull the body with iter_bytes / iter_lines (or aiter_bytes) so it never sits in memory

Where httpx differs from requests

The gotchas to remember when porting requests code.

PURPOSE	COMMAND	RESULT
Redirects are NOT followed by default.	`httpx.get(url,follow_redirects=True)`	
There is always a default timeout.	`httpx.get(url, timeout=` `  httpx.Timeout(10.0, connect=5.0))`	
Raw body uses content=, not data=.	`httpx.post(url,content=b"raw")`	
raise_for_status raises a different error.	`try: r.raise_for_status()` `except httpx.HTTPStatusError as e: ...`	
Check status without exceptions.	`r.is_success, r.is_error, r.is_redirect`	
Catch any request-side failure.	`except httpx.RequestError: # base: HTTPError`	

Four defaults bite when porting: no auto-redirect, a 5s timeout, content= for raw bytes, and httpx.HTTPStatusError instead of requests.HTTPError