

# asyncio

---

## Une troisième façon de faire de la concurrence

On a vu le threading et le multiprocessing. Il existe une troisième approche : **async I/O**, accessible via le module **asyncio** de la bibliothèque standard.

Ce qui la distingue : elle n'utilise ni plusieurs threads, ni plusieurs process. Tout se passe dans **un seul thread, un seul process**. Pourtant, elle permet d'exécuter plusieurs tâches "en même temps". Comment c'est possible ?

La réponse tient en un mot : **coopération**. Plutôt que de forcer l'alternance entre tâches (comme le fait le threading), asyncio laisse chaque tâche décider elle-même quand elle peut céder la main. On appelle ça la "coopérative multitasking".

## L'analogie du jeu d'échecs

Voici une image qui illustre parfaitement le principe.

Une joueuse d'échecs organise une exhibition : elle joue contre 24 adversaires en même temps.

Elle a deux façons de gérer ça :

**Version synchrone** : elle finit une partie complète avant de passer à la suivante. Chaque partie dure 30 minutes. L'exhibition entière dure  $24 \times 30 = 12$  heures.

**Version asynchrone** : elle fait le tour des tables, joue un coup à chaque table, et passe à la suivante pendant que l'adversaire réfléchit. Elle ne reste jamais à attendre. Résultat : l'exhibition entière dure **1 heure**.

Il n'y a pourtant qu'une seule joueuse. Elle ne joue pas vraiment "en parallèle", elle fait un coup à la fois. Mais elle exploite le temps d'attente des autres pour avancer partout.

C'est exactement ce que fait asyncio : pendant qu'une tâche attend (une réponse réseau, une lecture de fichier...), on passe la main à une autre tâche. Aucun CPU supplémentaire, aucun thread, juste une gestion intelligente des temps d'attente.

## async et await : les deux mots-clés

Tout asyncio repose sur deux mots-clés : **async** et **await**.

**async def** transforme une fonction ordinaire en **coroutine**. Une coroutine, c'est une fonction qui peut suspendre son exécution en cours de route et reprendre là où elle s'était arrêtée.

**await** est le point de suspension. Quand une coroutine rencontre un **await**, elle dit à l'event loop : "Je vais attendre ce résultat, profite-en pour lancer autre chose."

Voici la différence concrète entre une version synchrone et une version asynchrone du même code :

**Version synchrone :**

```
import time

def count():
    print("One")
    time.sleep(1)
    print("Two")
    time.sleep(1)

def main():
    for _ in range(3):
        count()

if __name__ == "__main__":
    start = time.perf_counter()
    main()
    elapsed = time.perf_counter() - start
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

Résultat :

```
One
Two
One
Two
One
Two
countsync.py executed in 6.03 seconds.
```

Les trois appels à `count()` se font l'un après l'autre. Chacun bloque pendant 2 secondes. Total : 6 secondes.

**Version asynchrone :**

```
import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")
    await asyncio.sleep(1)

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time

    start = time.perf_counter()
```

```

asyncio.run(main())
elapsed = time.perf_counter() - start
print(f"__file__ executed in {elapsed:0.2f} seconds.")

```

`asyncio.gather()` prend plusieurs coroutines en argument, les lance toutes en même temps sur l'event loop, et attend qu'elles soient **toutes** terminées avant de continuer. C'est la façon standard de lancer plusieurs tâches concurrentes. Il retourne une liste des résultats dans le même ordre que les coroutines passées en entrée.

`time.perf_counter()` donne un compteur de temps haute précision, mieux adapté que `time.time()` pour mesurer des durées courtes. Il retourne un nombre en secondes depuis un point de référence arbitraire, seule la différence entre deux appels a du sens.

Résultat :

```

One
One
One
Two
Two
Two
countasync.py executed in 2.00 seconds.

```

Les trois `count()` démarrent presque ensemble. Quand la première rencontre `await asyncio.sleep(1)`, elle laisse la main aux deux autres. Elles font toutes leur attente en même temps. Résultat : 2 secondes au lieu de 6.

Notez que `time.sleep()` est **bloquant** : il gèle tout le programme. `asyncio.sleep()` est **non-bloquant** : il suspend seulement la coroutine en cours et laisse l'event loop continuer. Dans du code async, utilisez toujours `asyncio.sleep()`.

## Les règles d'utilisation

Quelques règles à retenir pour éviter les erreurs de syntaxe :

```

async def f(x):
    y = await z(x)  # OK – await dans une coroutine
    return y

async def g(x):
    yield x  # OK – async generator

async def m(x):
    yield from gen(x)  # Erreur – yield from interdit dans async def

def n(x):

```

```
y = await z(x) # Erreur – await hors d'un async def
return y
```

En résumé : `await` ne s'utilise qu'à l'intérieur d'une fonction `async def`. Et on ne peut `await` que des objets "awaitables" : d'autres coroutines, ou des objets qui implémentent `__await__`.

## L'event loop

L'**event loop** est le chef d'orchestre d'asyncio. C'est lui qui tourne en permanence, surveille toutes les coroutines, et décide laquelle peut s'exécuter à chaque instant.

La façon moderne de le lancer :

```
asyncio.run(main())
```

`asyncio.run()` s'occupe de tout : démarrer l'event loop, exécuter la coroutine `main()` jusqu'au bout, puis fermer proprement. C'est la seule chose à retenir pour 95% des cas.

Une coroutine seule ne fait rien. Si vous faites `routine = main()`, vous obtenez un objet coroutine inerte. Il faut le passer à `asyncio.run()` pour qu'il s'exécute.

```
>>> import asyncio

>>> async def main():
...     print("Hello...")
...     await asyncio.sleep(1)
...     print("World!")
...

>>> routine = main()
>>> routine
<coroutine object main at 0x1027a6150>

>>> asyncio.run(routine)
Hello...
World!
```

## Le REPL asyncio

Depuis Python 3.8, vous pouvez tester du code async directement dans un shell interactif dédié :

```
$ python -m asyncio
```

Dans ce shell, `await` est utilisable directement sans `asyncio.run()` :

```
>>> import asyncio

>>> async def main():
...     print("Hello...")
...     await asyncio.sleep(1)
...     print("World!")
...

>>> await main()
Hello...
World!
```

Pratique pour expérimenter rapidement.

## Exemple plus complet : nombres aléatoires

Voici un exemple qui montre bien l'entrelacement des coroutines :

```
import asyncio
import random

COLORS = (
    "\033[0m", # End of color
    "\033[36m", # Cyan
    "\033[91m", # Red
    "\033[35m", # Magenta
)

async def makerandom(delay, threshold=6):
    color = COLORS[delay]
    print(f"{color}Initiated makerandom({delay}).")
    while (number := random.randint(0, 10)) <= threshold:
        print(f"{color}makerandom({delay}) == {number} too low; retrying.")
        await asyncio.sleep(delay)
    print(f"{color}---> Finished: makerandom({delay}) == {number}" +
COLORS[0])
    return number

async def main():
    return await asyncio.gather(
        makerandom(1, 9),
        makerandom(2, 8),
        makerandom(3, 8),
    )

if __name__ == "__main__":
    random.seed(444)
    r1, r2, r3 = asyncio.run(main())
```

```
print()
print(f"r1: {r1}, r2: {r2}, r3: {r3}")
```

Chaque appel à `makerandom()` tire un nombre aléatoire. Si le nombre est trop bas, il attend (avec son propre délai) et réessaie. Les trois instances tournent en même temps et s'entrelacent : pendant que l'une attend, les autres avancent. Les couleurs permettent de voir clairement quel appel fait quoi.

`random.seed(444)` fixe la graine du générateur aléatoire. Ça rend les tirages reproductibles : chaque fois que vous lancez le script avec la même graine, vous obtenez exactement les mêmes nombres. Utile pour les exemples pédagogiques et les tests.

`number := random.randint(0, 10)` utilise l'opérateur **walrus** (`:=`), disponible depuis Python 3.8. Il assigne et évalue en une seule expression : le résultat de `random.randint(0, 10)` est stocké dans `number` et testé dans la condition `while` en même temps. Sans lui, il faudrait deux lignes : d'abord l'assignation, puis le test.

## Enchaîner des coroutines (chaining)

Une coroutine peut `await` une autre coroutine. C'est ce qu'on appelle le **chaining** : on construit des pipelines où chaque étape attend le résultat de la précédente, tout en restant non-bloquant pour le reste du programme.

Exemple : récupérer des utilisateurs, puis pour chacun récupérer ses posts.

`random.uniform(a, b)` retourne un nombre décimal (float) aléatoire compris entre `a` et `b` inclus. Contrairement à `random.randint()` qui ne donne que des entiers, `uniform()` peut retourner n'importe quelle valeur à virgule dans l'intervalle : par exemple `random.uniform(0.5, 2.0)` pourrait donner `0.73`, `1.542` ou `1.99`. Ici on l'utilise pour simuler des délais réseau réalistes, qui ne tombent jamais pile sur des secondes entières.

```
import asyncio
import random
import time

async def fetch_posts(user):
    delay = random.uniform(0.5, 2.0)
    print(f"Post coro: retrieving posts for {user['name']}...")
    await asyncio.sleep(delay)
    posts = [f"Post {i} by {user['name']}" for i in range(1, 3)]
    print(
        f"Post coro: got {len(posts)} posts by {user['name']}"
        f" (done in {delay:.1f}s):"
    )
    for post in posts:
        print(f" - {post}")

async def fetch_user(user_id):
    delay = random.uniform(0.5, 2.0)
    print(f"User coro: fetching user by {user_id=}...")
    await asyncio.sleep(delay)
```

```

    user = {"id": user_id, "name": f"User{user_id}"}
    print(f"User coro: fetched user with {user_id=} (done in
{delay:.1f}s).")
    return user

async def get_user_with_posts(user_id):
    user = await fetch_user(user_id)
    await fetch_posts(user)

async def main():
    user_ids = [1, 2, 3]
    start = time.perf_counter()
    await asyncio.gather(
        *(get_user_with_posts(user_id) for user_id in user_ids) # dépack
le générateur en arguments séparés
    )
    end = time.perf_counter()
    print(f"\n==> Total time: {end - start:.2f} seconds")

if __name__ == "__main__":
    random.seed(444)
    asyncio.run(main())

```

Résultat :

```

User coro: fetching user by user_id=1...
User coro: fetching user by user_id=2...
User coro: fetching user by user_id=3...
User coro: fetched user with user_id=2 (done in 0.5s).
Post coro: retrieving posts for User2...
User coro: fetched user with user_id=1 (done in 1.0s).
Post coro: retrieving posts for User1...
User coro: fetched user with user_id=3 (done in 1.2s).
Post coro: retrieving posts for User3...
Post coro: got 2 posts by User2 (done in 1.8s):
  - Post 1 by User2
  - Post 2 by User2
Post coro: got 2 posts by User1 (done in 1.6s):
  - Post 1 by User1
  - Post 2 by User1
Post coro: got 2 posts by User3 (done in 1.5s):
  - Post 1 by User3
  - Post 2 by User3

==> Total time: 2.68 seconds

```

En synchrone, ce scénario prendrait ~7.6 secondes. Avec asyncio : 2.68 secondes. Les trois utilisateurs sont traités en parallèle, chacun avançant dès que sa coroutine peut reprendre la main.

La syntaxe `*(expression for x in liste)` dépacke un générateur en arguments séparés. `asyncio.gather()` attend des coroutines passées une par une, pas une liste. `*` transforme le générateur en autant d'arguments individuels. C'est équivalent à écrire `asyncio.gather(get_user_with_posts(1), get_user_with_posts(2), get_user_with_posts(3))` mais sans connaître la taille de la liste à l'avance.

## Producteurs et consommateurs avec une Queue

Le chaining fonctionne bien quand les étapes sont liées. Mais parfois on veut découpler complètement les producteurs (ceux qui génèrent des données) et les consommateurs (ceux qui les traitent).

`asyncio.Queue` est fait pour ça.

Le principe : on place une **queue** (file d'attente) entre les deux. Le producteur y dépose des éléments au fur et à mesure, sans se soucier de qui les traite. Le consommateur pioche dedans dès qu'un élément est disponible, sans savoir d'où il vient. Les deux avancent à leur propre rythme, de façon totalement indépendante.

Dans l'exemple qui suit, on a deux fonctions :

- **producer** : récupère des utilisateurs (simulé par un `asyncio.sleep`) et les pousse dans la queue un par un. Quand il a tout envoyé, il dépose des valeurs `None` dans la queue. Combien ? `len(user_ids)`, parce que dans cet exemple on lance exactement un consommateur par `user_id`. Chaque `None` sert de signal de fin pour un consommateur : "j'ai terminé, tu peux t'arrêter."
- **consumer** : tourne en boucle infinie et attend qu'un élément apparaisse dans la queue. Dès qu'il en reçoit un, il le traite (ici, récupérer les posts de l'utilisateur). S'il reçoit `None`, il sait que le producteur a fini et il s'arrête.

La fonction `main` se contente de créer la queue, lancer le producteur et les consommateurs en même temps avec `asyncio.gather()`, et mesurer le temps total.

```
import asyncio
import random
import time

async def producer(queue, user_ids):
    async def fetch_user(user_id):
        delay = random.uniform(0.5, 2.0)
        print(f"Producer: fetching user by {user_id}...")
        await asyncio.sleep(delay)
        user = {"id": user_id, "name": f"User{user_id}"}
        print(f"Producer: fetched user with {user_id} (done in {delay:.1f}s)")
        await queue.put(user)

    await asyncio.gather(*(fetch_user(uid) for uid in user_ids))
    for _ in range(len(user_ids)):
        await queue.put(None)

async def consumer(queue):
    while True:
```



```

        user = await queue.get()
        if user is None:
            break
        delay = random.uniform(0.5, 2.0)
        print(f"Consumer: retrieving posts for {user['name']}...")
        await asyncio.sleep(delay)
        posts = [f"Post {i} by {user['name']}" for i in range(1, 3)]
        print(
            f"Consumer: got {len(posts)} posts by {user['name']}"
            f" (done in {delay:.1f}s):"
        )
        for post in posts:
            print(f"  - {post}")

async def main():
    queue = asyncio.Queue()
    user_ids = [1, 2, 3]

    start = time.perf_counter()
    await asyncio.gather(
        producer(queue, user_ids),
        *(consumer(queue) for _ in user_ids),
    )
    end = time.perf_counter()
    print(f"\n==> Total time: {end - start:.2f} seconds")

if __name__ == "__main__":
    random.seed(444)
    asyncio.run(main())

```

`asyncio.Queue()` est une file d'attente thread-safe conçue pour `asyncio`. `await queue.put(item)` ajoute un élément, il suspend la coroutine si la queue est pleine (par défaut elle est illimitée). `await queue.get()` retire et retourne le prochain élément, il suspend la coroutine si la queue est vide, jusqu'à ce qu'un producteur ajoute quelque chose.

Les consommateurs n'ont aucune idée de combien d'éléments le producteur va générer, ils attendent juste que la queue soit alimentée. C'est un pattern très utilisé pour des pipelines de traitement de données.

## async for et async with

Python propose des versions asynchrones des constructs `for` et `with`.

**async for** s'utilise avec un itérateur asynchrone, un objet qui génère des données de façon non-bloquante :

```

import asyncio

async def powers_of_two(stop=10):
    exponent = 0
    while exponent < stop:

```

```

        yield 2**exponent
        exponent += 1
        await asyncio.sleep(0.2) # Simulate some asynchronous work

async def main():
    g = []
    async for i in powers_of_two(5):
        g.append(i)
    print(g)
    f = [j async for j in powers_of_two(5) if not (j // 3 % 5)]
    print(f)

asyncio.run(main())

```

`async for` ne rend pas l'itération concurrente en elle-même. Il permet juste à l'évent loop de continuer à tourner entre deux itérations, quand la coroutine fait un `await`. Si vous voulez vraiment itérer en parallèle, utilisez `asyncio.gather()`.

**async with** gère les ressources qui nécessitent une ouverture et une fermeture non-bloquantes. Exemple avec `aiohttp` pour vérifier si des sites sont en ligne :

```

import asyncio
import aiohttp

async def check(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            print(f"{url}: status -> {response.status}")

async def main():
    websites = [
        "https://realpython.com",
        "https://pycoders.com",
        "https://www.python.org",
    ]
    await asyncio.gather(*(check(url) for url in websites))

asyncio.run(main())

```

`async with` garantit que les connexions réseau sont ouvertes et fermées proprement, même si une erreur survient. `aiohttp` s'installe avec `pip install aiohttp`.

## Les outils principaux d'asyncio

`asyncio.gather()`, lance plusieurs coroutines en même temps et attend qu'elles soient toutes terminées. Retourne les résultats dans l'ordre dans lequel les coroutines ont été passées.

`asyncio.create_task()`, planifie l'exécution d'une coroutine sur l'évent loop sans attendre immédiatement son résultat. Retourne un objet `Task` que vous pouvez `await` plus tard.

```
import asyncio

async def coro(numbers):
    await asyncio.sleep(min(numbers))
    return list(reversed(numbers))

async def main():
    task = asyncio.create_task(coro([3, 2, 1]))
    print(f"{type(task) = }")
    print(f"{task.done() = }")
    return await task

result = asyncio.run(main())
```

Important : si vous créez des tasks avec `create_task()` mais ne les `await` pas, elles seront annulées quand l'event loop se ferme. Toujours `await` vos tasks.

La différence entre `create_task()` et `await` directement :

- `await ma_coroutine()` : exécute la coroutine immédiatement et attend qu'elle finisse avant de continuer
- `asyncio.create_task(ma_coroutine())` : planifie la coroutine en arrière-plan, vous pouvez continuer à faire autre chose et `await` le résultat plus tard

`asyncio.as_completed()`, vous donne les résultats des tasks au fur et à mesure qu'elles se terminent, pas dans l'ordre de démarrage :

```
import asyncio, time

async def coro(numbers):
    await asyncio.sleep(min(numbers))
    return list(reversed(numbers))

async def main():
    task1 = asyncio.create_task(coro([10, 5, 2]))
    task2 = asyncio.create_task(coro([3, 2, 1]))
    print("Start:", time.strftime("%X"))
    for task in asyncio.as_completed([task1, task2]):
        result = await task
        print(f'result: {result} completed at {time.strftime("%X")}')
    print("End:", time.strftime("%X"))
    print(f"Both tasks done: {all((task1.done(), task2.done()))}")

asyncio.run(main())
```

```
Start: 14:36:36
result: [1, 2, 3] completed at 14:36:37
result: [2, 5, 10] completed at 14:36:38
```

```
End: 14:36:38
Both tasks done: True
```

Utile quand vous voulez traiter les résultats dès qu'ils arrivent, sans attendre les plus lents.

## Gérer les exceptions de plusieurs coroutines (Python 3.11+)

Quand plusieurs coroutines tournent en même temps, plusieurs peuvent lever des exceptions. Depuis Python 3.11, `ExceptionGroup` et le mot-clé `except*` permettent de toutes les attraper proprement :

```
import asyncio

async def coro_a():
    await asyncio.sleep(1)
    raise ValueError("Error in coro A")

async def coro_b():
    await asyncio.sleep(2)
    raise TypeError("Error in coro B")

async def coro_c():
    await asyncio.sleep(0.5)
    raise IndexError("Error in coro C")

async def main():
    results = await asyncio.gather(
        coro_a(),
        coro_b(),
        coro_c(),
        return_exceptions=True
    )
    exceptions = [e for e in results if isinstance(e, Exception)]
    if exceptions:
        raise ExceptionGroup("Errors", exceptions)

try:
    asyncio.run(main())
except* ValueError as ve_group:
    print(f"[ValueError handled] {ve_group.exceptions}")
except* TypeError as te_group:
    print(f"[TypeError handled] {te_group.exceptions}")
except* IndexError as ie_group:
    print(f"[IndexError handled] {ie_group.exceptions}")
```

```
[ValueError handled] (ValueError('Error in coro A'),)
[TypeError handled] (TypeError('Error in coro B'),)
[IndexError handled] (IndexError('Error in coro C'),)
```

`return_exceptions=True` dans `gather()` empêche une exception d'annuler toutes les autres coroutines, elles s'exécutent jusqu'au bout et les exceptions sont retournées comme résultats.

## Les pièges classiques

async ne rend pas votre code magiquement plus rapide. Ce qui suit, c'est une liste d'erreurs réelles, des équipes entières s'y sont fait prendre en production.

### Piège 1 : mettre async def sans changer les appels bloquants

C'est l'erreur la plus fréquente. Un développeur a migré toute une API FastAPI de `def` vers `async def` un week-end. Résultat : le temps de réponse est passé de 180ms à 1400ms. Voici pourquoi.

Dans les exemples ci-dessous, `get_db` est une dépendance FastAPI classique qui ouvre une session SQLAlchemy et la ferme proprement après chaque requête. `Depends(get_db)` dit à FastAPI : "injecte-moi une session de base de données dans ce paramètre".

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, Session

engine = create_engine("postgresql://user:password@localhost/mydb")
SessionLocal = sessionmaker(bind=engine)

def get_db():
    db = SessionLocal()
    try:
        yield db          # FastAPI injecte cette session dans l'endpoint
    finally:
        db.close()       # exécuté même si l'endpoint lève une exception
```

```
# Version sync, fonctionne correctement
@app.get("/users/{user_id}")
def get_user(user_id: int, db: Session = Depends(get_db)):
    user = db.query(User).filter(User.id == user_id).first()
    return user

# Version "async", en réalité pire
@app.get("/users/{user_id}")
async def get_user(user_id: int, db: Session = Depends(get_db)):
    user = db.query(User).filter(User.id == user_id).first()
    return user
```

Le seul changement : `def` → `async def`. Mais `db.query()` est toujours synchrone.

Ce qui se passe avec `def` : FastAPI exécute la fonction dans un thread pool. Pendant que ce thread attend la base de données, d'autres threads traitent d'autres requêtes. L'event loop reste libre.

Ce qui se passe avec `async def` : FastAPI exécute la fonction directement sur l'event loop. Quand `db.query()` bloque en attendant la base de données, il bloque **tout l'event loop**. Aucune autre requête ne peut passer. Les utilisateurs font la queue les uns derrière les autres.

**Règle** : si votre fonction appelle quelque chose de bloquant (SQLAlchemy sync, `requests`, `time.sleep()`, lecture de fichier sans `aiofiles`), utilisez `def`, pas `async def`. Le thread pool de FastAPI s'occupe de la concurrence.

Piège 2 : utiliser `requests` dans du code async

```
import requests

# Bloque l'event loop pendant 200-500ms
@app.get("/weather")
async def get_weather(city: str):
    response = requests.get(f"https://api.weather.com/{city}")
    return response.json()
```

`requests` est synchrone. Il bloque pendant toute la durée de la requête HTTP. À l'intérieur d'une `async def`, ça immobilise l'event loop.

La solution : `httpx`, qui a exactement la même interface mais supporte l'async :

```
import httpx

@app.get("/weather")
async def get_weather(city: str):
    async with httpx.AsyncClient() as client:
        response = await client.get(f"https://api.weather.com/{city}")
        return response.json()
```

En production, évitez de créer un nouveau client à chaque requête (chaque création implique une négociation TCP + TLS). Partagez un client au niveau de l'application :

`@asynccontextmanager` est un decorator de la bibliothèque standard (`contextlib`) qui transforme une fonction génératrice async en context manager utilisable avec `async with`. Tout ce qui est avant `yield` s'exécute à l'entrée du bloc `async with`, et tout ce qui est après `yield` s'exécute à la sortie, même si une exception est levée. C'est la façon propre de gérer des ressources (connexions, fichiers, clients HTTP) qui doivent être ouvertes puis fermées.

```
from contextlib import asynccontextmanager
import httpx

@asynccontextmanager
async def lifespan(app: FastAPI):
    app.state.http_client = httpx.AsyncClient(timeout=10.0)
    yield
```

```

        await app.state.http_client.aclose()

app = FastAPI(lifespan=lifespan)

@app.get("/weather")
async def get_weather(city: str, request: Request):
    client = request.app.state.http_client
    response = await client.get(f"https://api.weather.com/{city}")
    return response.json()

```

Un client partagé avec réutilisation des connexions peut couper les temps de réponse de 30 à 40% sur les endpoints qui appellent des APIs externes.

### Piège 3 : faire du calcul CPU sur l'event loop

```

import hashlib

@app.post("/process")
async def process_file(file: UploadFile):
    content = await file.read()

    # Calcul CPU, bloque l'event loop
    hash_result = hashlib.sha256(content).hexdigest()
    processed = heavy_data_processing(content)

    return {"hash": hash_result, "size": len(processed)}

```

`await file.read()` est correct. Mais `hashlib.sha256()` et `heavy_data_processing()` tournent sur le thread de l'event loop et bloquent tout pendant leur exécution.

Pour du travail CPU lourd, délégez-le à un executor :

```

import asyncio
from concurrent.futures import ProcessPoolExecutor

executor = ProcessPoolExecutor(max_workers=4)

def heavy_processing(content: bytes):
    """Tourne dans un process séparé, ne bloque pas l'event loop."""
    hash_result = hashlib.sha256(content).hexdigest()
    processed = heavy_data_processing(content)
    return hash_result, processed

@app.post("/process")
async def process_file(file: UploadFile):
    content = await file.read()

    loop = asyncio.get_event_loop()
    hash_result, processed = await loop.run_in_executor(

```

```

        executor, heavy_processing, content
    )

    return {"hash": hash_result, "size": len(processed)}

```

`loop.run_in_executor(executor, fonction, *args)` exécute `fonction(*args)` dans l'executor (thread pool ou process pool) et retourne un awaitable. L'event loop reste libre pendant toute la durée de l'exécution. `asyncio.get_event_loop()` récupère l'instance de l'event loop qui tourne dans le thread courant.

Règle : `ThreadPoolExecutor` pour du I/O bloquant qu'on ne peut pas rendre async (bibliothèques legacy). `ProcessPoolExecutor` pour du vrai calcul CPU.

#### Piège 4 : mélanger sessions sync et async

Avec SQLAlchemy, il existe une session synchrone et une session asynchrone. Les mélanger dans le même projet est un terrain miné.

Voici à quoi ressemble la version async de `get_db` :

```

from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.orm import sessionmaker

async_engine =
create_async_engine("postgresql+asyncpg://user:password@localhost/mydb")
async_session_maker = sessionmaker(async_engine, class_=AsyncSession,
expire_on_commit=False)

async def get_async_db():
    async with async_session_maker() as session:
        yield session

```

`asyncpg` est le driver PostgreSQL async — sans lui, même avec `AsyncSession`, les appels resteraient bloquants. Chaque base de données a son propre driver async (`aiomysql` pour MySQL, `aiosqlite` pour SQLite...).

```

# DANGEREUX, session sync dans un endpoint async
@app.get("/users")
async def list_users(db: Session = Depends(get_db)):
    users = db.query(User).all() # Bloque l'event loop !
    return users

# CORRECT, session async dans un endpoint async
@app.get("/users")
async def list_users(db: AsyncSession = Depends(get_async_db)):
    result = await db.execute(select(User))
    return result.scalars().all()

```



```
# AUSSI CORRECT, session sync dans un endpoint sync
@app.get("/users")
def list_users(db: Session = Depends(get_db)):
    return db.query(User).all() # FastAPI l'exécute dans un thread pool
```

La recommandation pour la plupart des projets : choisissez une approche et tenez-vous-y. Pour du CRUD standard avec SQLAlchemy, les sessions sync avec `def` sont suffisantes et bien plus simples à maintenir.

### Piège 5 : oublier le `await`

```
@app.get("/data")
async def get_data():
    result = fetch_data_async() # await manquant !
    return {"data": result}
```

Si `fetch_data_async` est une coroutine et que vous oubliez `await`, vous ne récupérez pas la donnée. Vous récupérez l'objet coroutine. Python ne lève pas d'erreur, votre endpoint retourne silencieusement quelque chose comme `{"data": "<coroutine object fetch_data_async at 0x7f...>"}`.

Python émet un `RuntimeWarning: coroutine 'fetch_data_async' was never awaited`, mais si vos logs ne remontent pas les warnings, vous ne le voyez pas. Les linters modernes l'attrapent la plupart du temps, mais pas toujours dans du code complexe.

### Piège 6 : tout passer en async "par précaution"

Après s'être fait brûler par un problème async, certains font l'inverse : ils rendent tout async "pour être prêts". C'est une autre erreur.

Du code async est plus difficile à déboguer. Les stack traces sont moins lisibles. La gestion des exceptions se comporte différemment. Les tests nécessitent `pytest-asyncio`. Chaque bibliothèque de votre stack doit avoir une version async compatible.

Les benchmarks sur une API CRUD standard montrent que la version async correcte est ~13% plus rapide que la version sync. Mais la version mal faite (async def avec des appels bloquants) est 80% plus lente.

Pour la grande majorité des APIs web, les endpoints `def` avec le thread pool de FastAPI sont largement suffisants.

## Quand utiliser asyncio ?

La question à se poser pour chaque endpoint :

- L'endpoint utilise des bibliothèques synchrones (SQLAlchemy sync, `requests`, `boto3`...) ? → `def`, laissez le thread pool travailler.
- L'endpoint utilise uniquement des bibliothèques async (`httpx`, SQLAlchemy async, `aiofiles`...) ? → `async def` avec `await` sur chaque appel I/O.

- L'endpoint fait du calcul CPU lourd ? → **def** ou déléguez à **run\_in\_executor** avec **ProcessPoolExecutor**.
- C'est un endpoint WebSocket ? → forcément **async def**.
- Vous n'êtes pas sûr ? → **def**. C'est toujours safe. Vous optimiserez si le profiling le justifie.

## Bibliothèques à connaître

L'écosystème asyncio est très riche. Voici les incontournables :

### Frameworks web :

- **FastAPI**, API REST async, très utilisé en production
- **Starlette**, le socle sur lequel FastAPI est construit
- **Quart**, même API que Flask, mais async

### Serveurs ASGI :

- **uvicorn**, serveur rapide pour les apps async

### HTTP et réseau :

- **aiohttp**, client et serveur HTTP async
- **HTTPX**, client HTTP avec support async et sync
- **websockets**, WebSocket avec asyncio

### Bases de données :

- **Databases**, accès async compatible SQLAlchemy
- **Motor**, driver async pour MongoDB

### Utilitaires :

- **aiofiles**, lecture/écriture de fichiers en async
- **pytest-asyncio**, pour tester vos fonctions async avec pytest

Vous avez ces trois fonctions synchrones qui simulent des appels réseau lents :

```
import time

def fetch_users():
    time.sleep(2)
    return ["alice", "bob", "charlie"]

def fetch_orders():
    time.sleep(3)
    return ["order_1", "order_2"]

def fetch_inventory():
    time.sleep(1)
    return ["item_A", "item_B", "item_C"]
```

1. Écrivez une version synchrone qui appelle les trois fonctions l'une après l'autre et mesure le temps total.
2. Convertissez les trois fonctions en coroutines (`asyncio.sleep` à la place de `time.sleep`).
3. Écrivez une version asynchrone qui les lance toutes les trois en même temps avec `asyncio.gather()` et mesure le temps total.
4. Comparez les deux durées et expliquez pourquoi elles diffèrent.

Résultat attendu pour la version async : environ 3 secondes (le maximum des trois), pas 6 (la somme).

## Exercice 2 — Téléchargeur concurrent avec progression

Simulez le téléchargement de 5 fichiers en parallèle. Chaque fichier a une taille aléatoire entre 1 et 5 secondes de téléchargement.

Contraintes :

- Utiliser `asyncio.create_task()` pour lancer les téléchargements
- Afficher un message dès qu'un fichier est terminé, dans l'ordre d'achèvement (pas l'ordre de lancement) — utilisez `asyncio.as_completed()`
- Afficher à la fin le temps total et le nom du fichier le plus long à télécharger

Exemple de sortie attendue :

```
Début des téléchargements...
✓ fichier_3.zip terminé (1.2s)
✓ fichier_1.zip terminé (2.4s)
✓ fichier_5.zip terminé (3.1s)
✓ fichier_2.zip terminé (4.0s)
✓ fichier_4.zip terminé (4.8s)
Temps total : 4.8s – fichier le plus lent : fichier_4.zip
```

## Exercice 3 — Timeout sur des tâches réseau

Vous interrogez 4 serveurs. Certains répondent vite, d'autres sont lents ou ne répondent jamais. Simulez ce comportement :

```
SERVEURS = {
    "serveur_A": 0.5,
    "serveur_B": 2.0,
    "serveur_C": 5.0,    # trop lent
    "serveur_D": 1.2,
}
```

Écrivez une coroutine `interroger(nom, delai)` qui simule l'appel au serveur. Ensuite, dans `main()` :

1. Appliquez un timeout de 2 secondes à chaque appel avec `asyncio.wait_for()`
2. Si un serveur dépasse le timeout, affichez un message d'avertissement et continuez avec les autres

3. Affichez à la fin un résumé : serveurs ayant répondu / serveurs en timeout

### Exercice 4 — Pipeline producteur / consommateur

Vous avez une liste de 6 URLs d'API à interroger (simulées). Chaque URL retourne une liste de résultats qui doit ensuite être analysée.

Mettez en place un pipeline avec `asyncio.Queue` :

- 1 producteur qui "récupère" les données de chaque URL (délai aléatoire 0.5–2s)
- 2 consommateurs qui analysent les données en parallèle dès qu'elles arrivent (délai fixe 1s par analyse)
- Le producteur signale la fin avec des sentinelles `None`
- Affichez le temps total et comparez-le avec ce que prendrait un pipeline séquentiel

L'objectif : montrer que 2 consommateurs en parallèle traitent plus vite qu'un seul.

### Exercice 5 — Détecteur de services actifs

Écrivez un script `check_services.py` qui vérifie si une liste de services locaux répond, en simulant des checks de disponibilité :

```
SERVICES = [  
    {"nom": "base de données", "delai": 0.3},  
    {"nom": "cache Redis",      "delai": 0.1},  
    {"nom": "API externe",      "delai": 3.5}, # timeout  
    {"nom": "serveur mail",     "delai": 0.8},  
    {"nom": "CDN",              "delai": 1.5},  
]
```

Contraintes :

- Tous les checks se lancent en même temps
- Timeout de 2 secondes par service
- Résultat affiché dès qu'un check est terminé (pas à la fin de tous)
- Un service en timeout est marqué **HORS LIGNE**
- À la fin : nombre de services en ligne / hors ligne, et temps total

Ce script doit ressembler à un vrai outil de monitoring : propre, lisible, utilisable en production.