

Concurrence et les mots-clés `async` et `await`

Cette page vise à fournir des détails sur la syntaxe `async def` pour les *fonctions de chemins* et quelques rappels sur le code asynchrone, la concurrence et le parallélisme.

Vous êtes pressés ?

TL;DR:

Si vous utilisez des bibliothèques tierces qui nécessitent d'être appelées avec `await`, telles que :

```
results = await some_library()
```

Alors, déclarez vos *fonctions de chemins* avec `async def` comme ceci :

```
@app.get('/')
async def read_results():
    results = await some_library()
    return results
```

!!! note Vous pouvez uniquement utiliser `await` dans les fonctions créées avec `async def`.

Si vous utilisez une bibliothèque externe qui communique avec quelque chose (une BDD, une API, un système de fichiers, etc.) et qui ne supporte pas l'utilisation de `await` (ce qui est actuellement le cas pour la majorité des bibliothèques de BDD), alors déclarez vos *fonctions de chemin* normalement, avec le classique `def`, comme ceci :

```
@app.get('/')
def results():
    results = some_library()
    return results
```

Si votre application n'a pas à communiquer avec une bibliothèque externe et pas à attendre de réponse, utilisez `async def`.

Si vous ne savez pas, utilisez seulement `def` comme vous le feriez habituellement.

Note : vous pouvez mélanger `def` et `async def` dans vos *fonctions de chemin* autant que nécessaire, **FastAPI** saura faire ce qu'il faut avec.

Au final, peu importe le cas parmi ceux ci-dessus, **FastAPI** fonctionnera de manière asynchrone et sera extrêmement rapide.

Mais si vous suivez bien les instructions ci-dessus, alors **FastAPI** pourra effectuer quelques optimisations et ainsi améliorer les performances.





Détails techniques


Les versions modernes de Python supportent le **code asynchrone** grâce aux "**coroutines**" avec les syntaxes `async` et `await`.




Analysons les différentes parties de cette phrase dans les sections suivantes :



- **Code asynchrone**
- `async` et `await`
- **Coroutines**

Code asynchrone

Faire du code asynchrone signifie que le langage  est capable de dire à l'ordinateur / au programme  qu'à un moment du code, il  devra attendre que *quelque chose d'autre* se termine autre part. Disons que ce *quelque chose d'autre* est appelé "fichier-lent" .

Donc, pendant ce temps, l'ordinateur pourra effectuer d'autres tâches, pendant que "fichier-lent"  se termine.

Ensuite l'ordinateur / le programme  reviendra à chaque fois qu'il en a la chance que ce soit parce qu'il attend à nouveau, ou car il  a fini tout le travail qu'il avait à faire. Il  regardera donc si les tâches qu'il attend ont terminé d'être effectuées.

Ensuite, il  prendra la première tâche à finir (disons, notre "fichier-lent" ) et continuera à faire avec cette dernière ce qu'il était censé.

Ce "attendre quelque chose d'autre" fait généralement référence à des opérations I/O qui sont relativement "lentes" (comparées à la vitesse du processeur et de la mémoire RAM) telles qu'attendre que :

- de la donnée soit envoyée par le client à travers le réseau
- de la donnée envoyée depuis votre programme soit reçue par le client à travers le réseau
- le contenu d'un fichier sur le disque soit lu par le système et passé à votre programme
- le contenu que votre programme a passé au système soit écrit sur le disque
- une opération effectuée à distance par une API se termine
- une opération en BDD se termine
- une requête à une BDD renvoie un résultat
- etc.

Le temps d'exécution étant consommé majoritairement par l'attente d'opérations I/O on appelle ceci des opérations "I/O bound".

Ce concept se nomme l'"asynchronisme" car l'ordinateur / le programme n'a pas besoin d'être "synchronisé" avec la tâche, attendant le moment exact où cette dernière se terminera en ne faisant rien, pour être capable de récupérer le résultat de la tâche et l'utiliser dans la suite des opérations.

À la place, en étant "asynchrone", une fois terminée, une tâche peut légèrement attendre (quelques microsecondes) que l'ordinateur / le programme finisse ce qu'il était en train de faire, et revienne récupérer le résultat.

Pour parler de tâches "synchrones" (en opposition à "asynchrones"), on utilise souvent le terme "séquentiel", car l'ordinateur / le programme va effectuer toutes les étapes d'une tâche séquentiellement avant de passer à une autre tâche, même si ces étapes impliquent de l'attente.

Concurrence et Burgers

L'idée de code **asynchrone** décrite ci-dessus est parfois aussi appelée "**concurrence**". Ce qui est différent du "**parallélisme**".

La **concurrence** et le **parallélisme** sont tous deux liés à l'idée de "différentes choses arrivant plus ou moins au même moment".

Mais les détails entre la **concurrence** et le **parallélisme** diffèrent sur de nombreux points.

Pour expliquer la différence, voici une histoire de burgers :

Burgers concurrents

Vous amenez votre crush 🥰 dans votre fast food 🍔 favori, et faites la queue pendant que le serveur 🧑🍳 prend les commandes des personnes devant vous.

Puis vient votre tour, vous commandez alors 2 magnifiques burgers 🍔 pour votre crush 🥰 et vous.

Vous payez 💰.

Le serveur 🧑🍳 dit quelque chose à son collègue dans la cuisine 👨🍳 pour qu'il sache qu'il doit préparer vos burgers 🍔 (bien qu'il soit déjà en train de préparer ceux des clients précédents).

Le serveur 🧑🍳 vous donne le numéro assigné à votre commande.

Pendant que vous attendez, vous allez choisir une table avec votre crush 🥰, vous discutez avec votre crush 🥰 pendant un long moment (les burgers étant "magnifiques" ils sont très longs à préparer 🌟🍔🌟).

Pendant que vous êtes assis à table, en attendant que les burgers 🍔 soient prêts, vous pouvez passer ce temps à admirer à quel point votre crush 🥰 est géniale, mignonne et intelligente 🌟🥰🌟.

Pendant que vous discutez avec votre crush 🥰, de temps en temps vous jetez un coup d'oeil au nombre affiché au-dessus du comptoir pour savoir si c'est à votre tour d'être servis.

Jusqu'au moment où c'est (enfin) votre tour. Vous allez au comptoir, récupérez vos burgers 🍔 et revenez à votre table.

Vous et votre crush 🥰 mangez les burgers 🍔 et passez un bon moment 🌟.

Imaginez que vous êtes l'ordinateur / le programme 🖥️ dans cette histoire.

Pendant que vous faites la queue, vous êtes simplement inactif 🤡, attendant votre tour, ne faisant rien de "productif". Mais la queue est rapide car le serveur 🧑🍳 prend seulement les commandes (et ne les prépare pas), donc tout va bien.

Ensuite, quand c'est votre tour, vous faites des actions "productives" 🧐, vous étudiez le menu, décidez ce que vous voulez, demandez à votre crush 🥰 son choix, payez 💰, vérifiez que vous utilisez la bonne carte de crédit, vérifiez que le montant débité sur la carte est correct, vérifiez que la commande contient les bons produits, etc.

Mais ensuite, même si vous n'avez pas encore vos burgers 🍔, votre travail avec le serveur 🧑🍳 est "en pause" ⏸️, car vous devez attendre ⌚ que vos burgers soient prêts.

Après vous être écarté du comptoir et vous être assis à votre table avec le numéro de votre commande, vous pouvez tourner 🔄 votre attention vers votre crush 🥰, et "travailler" 💻 🧐 là-dessus. Vous êtes donc à nouveau en train de faire quelque chose de "productif" 🧐, vous flirtez avec votre crush 🥰.

Puis le serveur 🧑🍳 dit "J'ai fini de préparer les burgers" 🍔 en mettant votre numéro sur l'affichage du comptoir, mais vous ne courez pas immédiatement au moment où votre numéro s'affiche. Vous savez que personne ne volera vos burgers 🍔 car vous avez votre numéro et les autres clients ont le leur.

Vous attendez donc que votre crush 🧑🏻💖 finisse son histoire, souriez gentiment et dites que vous allez chercher les burgers 🍔.

Pour finir vous allez au comptoir 🧑🏻💻, vers la tâche initiale qui est désormais terminée 🟩, récupérez les burgers 🍔, remerciez le serveur et ramenez les burgers 🍔 à votre table. Ceci termine l'étape / la tâche d'interaction avec le comptoir 🟩. Ce qui ensuite, crée une nouvelle tâche de "manger les burgers" 🧑🏻💻 🍔, mais la précédente, "récupérer les burgers" est terminée 🟩.

Burgers parallèles

Imaginons désormais que ce ne sont pas des "burgers concurrents" mais des "burgers parallèles".

Vous allez avec votre crush 🧑🏻💖 dans un fast food 🍔 parallélisé.

Vous attendez pendant que plusieurs (disons 8) serveurs qui sont aussi des cuisiniers 🧑🏻👩🍳🧑🏻👩🍳🧑🏻👩🍳🧑🏻👩🍳 prennent les commandes des personnes devant vous.

Chaque personne devant vous attend ⌚ que son burger 🍔 soit prêt avant de quitter le comptoir car chacun des 8 serveurs va lui-même préparer le burger directement avant de prendre la commande suivante.

Puis c'est enfin votre tour, vous commandez 2 magnifiques burgers 🍔 pour vous et votre crush 🧑🏻💖.

Vous payez 💵.

Le serveur va dans la cuisine 🧑🏻👩🍳.

Vous attendez devant le comptoir afin que personne ne prenne vos burgers 🍔 avant vous, vu qu'il n'y a pas de numéro de commande.

Vous et votre crush 🧑🏻💖 étant occupés à vérifier que personne ne passe devant vous prendre vos burgers au moment où ils arriveront ⌚, vous ne pouvez pas vous préoccuper de votre crush 🧑🏻💖.

C'est du travail "synchrone", vous êtes "synchronisés" avec le serveur/cuisinier 🧑🏻👩🍳. Vous devez attendre ⌚ et être présent au moment exact où le serveur/cuisinier 🧑🏻👩🍳 finira les burgers 🍔 et vous les donnera, sinon quelqu'un risque de vous les prendre.

Puis le serveur/cuisinier 🧑🏻👩🍳 revient enfin avec vos burgers 🍔, après un long moment d'attente ⌚ devant le comptoir.

Vous prenez vos burgers 🍔 et allez à une table avec votre crush 🧑🏻💖

Vous les mangez, et vous avez terminé 🍔 🟩.

Durant tout ce processus, il n'y a presque pas eu de discussions ou de flirts car la plupart de votre temps a été passé à attendre ⌚ devant le comptoir 🧑🏻💻.

Dans ce scénario de burgers parallèles, vous êtes un ordinateur / programme 🧑🏻💻 avec deux processeurs (vous et votre crush 🧑🏻💖) attendant ⌚ à deux et dédiant votre attention ⌚ à "attendre devant le comptoir" pour une longue durée.

Le fast-food a 8 processeurs (serveurs/cuisiniers) 🧑🏻👩🍳🧑🏻👩🍳🧑🏻👩🍳🧑🏻👩🍳. Alors que le fast-food de burgers concurrents en avait 2 (un serveur et un cuisinier).

Et pourtant l'expérience finale n'est pas meilleure 🧑🏻💖.

C'est donc l'histoire équivalente parallèle pour les burgers 🍔.

Pour un exemple plus courant dans la "vie réelle", imaginez une banque.

Jusqu'à récemment, la plupart des banques avaient plusieurs caisses (et banquiers) 🧑🧑🧑🧑 et une unique file d'attente ⌚⌚⌚⌚⌚⌚⌚⌚.

Tous les banquiers faisaient l'intégralité du travail avec chaque client avant de passer au suivant 🧑🏦.

Et vous deviez attendre ⌚ dans la file pendant un long moment ou vous perdiez votre place.

Vous n'auriez donc probablement pas envie d'amener votre crush 🥰 avec vous à la banque 🏦.

Conclusion

Dans ce scénario des "burgers du fast-food avec votre crush", comme il y a beaucoup d'attente ⌚, il est très logique d'avoir un système concurrent 🏠🔗🏠.

Et c'est le cas pour la plupart des applications web.

Vous aurez de nombreux, nombreux utilisateurs, mais votre serveur attendra ⌚ que leur connexion peu performante envoie des requêtes.

Puis vous attendrez ⌚ de nouveau que leurs réponses reviennent.

Cette "attente" ⌚ se mesure en microsecondes, mais tout de même, en cumulé cela fait beaucoup d'attente.

C'est pourquoi il est logique d'utiliser du code asynchrone 🏠🔗🏠 pour des APIs web.

La plupart des frameworks Python existants (y compris Flask et Django) ont été créés avant que les nouvelles fonctionnalités asynchrones de Python n'existent. Donc, les façons dont ils peuvent être déployés supportent l'exécution parallèle et une ancienne forme d'exécution asynchrone qui n'est pas aussi puissante que les nouvelles fonctionnalités de Python.

Et cela, bien que les spécifications principales du web asynchrone en Python (ou ASGI) ont été développées chez Django, pour ajouter le support des WebSockets.

Ce type d'asynchronicité est ce qui a rendu NodeJS populaire (bien que NodeJS ne soit pas parallèle) et c'est la force du Go en tant que langage de programmation.

Et c'est le même niveau de performance que celui obtenu avec **FastAPI**.

Et comme on peut avoir du parallélisme et de l'asynchronicité en même temps, on obtient des performances plus hautes que la plupart des frameworks NodeJS et égales à celles du Go, qui est un langage compilé plus proche du C ([tout ça grâce à Starlette](#)).

Est-ce que la concurrence est mieux que le parallélisme ?

Nope ! C'est ça la morale de l'histoire.

La concurrence est différente du parallélisme. C'est mieux sur des scénarios **spécifiques** qui impliquent beaucoup d'attente. À cause de ça, c'est généralement bien meilleur que le parallélisme pour le développement d'applications web. Mais pas pour tout.

Donc pour équilibrer tout ça, imaginez l'histoire suivante :

Vous devez nettoyer une grande et sale maison.

Oui, c'est toute l'histoire.

Il n'y a plus d'attente ⌚ nulle part, juste beaucoup de travail à effectuer, dans différentes pièces de la maison.

Vous pourriez diviser en différentes sections comme avec les burgers, d'abord le salon, puis la cuisine, etc. Mais vous n'attendez ⌚ rien, vous ne faites que nettoyer et nettoyer, la séparation en sections ne changerait rien au final.

Cela prendrait autant de temps pour finir avec ou sans sections (concurrency) et vous auriez effectué la même quantité de travail.

Mais dans ce cas, si pouviez amener 8 ex-serveurs/cuisiniers/devenus-nettoyeurs 🧑🍳🧑🍳🧑🍳🧑🍳🧑🍳🧑🍳🧑🍳, et que chacun d'eux (plus vous) pouvait prendre une zone de la maison pour la nettoyer, vous pourriez faire tout le travail en parallèle, et finir plus tôt.

Dans ce scénario, chacun des nettoyeurs (vous y compris) serait un processeur, faisant sa partie du travail.

Et comme la plupart du temps d'exécution est pris par du "vrai" travail (et non de l'attente), et que le travail dans un ordinateur est fait par un CPU, ce sont des problèmes dits "CPU bound".

Des exemples communs d'opérations "CPU bounds" sont les procédés qui requièrent des traitements mathématiques complexes.

Par exemple :

- Traitements d'**audio** et d'**images**.
- La **vision par ordinateur** : une image est composée de millions de pixels, chaque pixel ayant 3 valeurs / couleurs, les traiter tous va nécessiter d'effectuer des traitements sur chaque pixel, et de préférence tous en même temps.
- L'apprentissage automatique (ou **Machine Learning**) : cela nécessite de nombreuses multiplications de matrices et vecteurs. Imaginez une énorme feuille de calcul remplie de nombres que vous multipliez entre eux tous au même moment.
- L'apprentissage profond (ou **Deep Learning**) : est un sous-domaine du **Machine Learning**, donc les mêmes raisons s'appliquent. Avec la différence qu'il n'y a pas une unique feuille de calcul de nombres à multiplier, mais une énorme quantité d'entre elles, et dans de nombreux cas, on utilise un processeur spécial pour construire et / ou utiliser ces modèles.

Concurrence + Parallélisme : Web + Machine Learning

Avec **FastAPI** vous pouvez bénéficier de la concurrence qui est très courante en développement web (c'est l'attrait principal de NodeJS).

Mais vous pouvez aussi profiter du parallélisme et multiprocessing afin de gérer des charges **CPU bound** qui sont récurrentes dans les systèmes de *Machine Learning*.

Ça, ajouté au fait que Python soit le langage le plus populaire pour la **Data Science**, le **Machine Learning** et surtout le **Deep Learning**, font de **FastAPI** un très bon choix pour les APIs et applications de **Data Science / Machine Learning**.





Pour comprendre comment mettre en place ce parallélisme en production, allez lire la section [Déploiement](#){internal-link target=_blank}.

async et await

Les versions modernes de Python ont une manière très intuitive de définir le code asynchrone, tout en gardant une apparence de code "séquentiel" classique en laissant Python faire l'attente pour vous au bon moment.

Pour une opération qui nécessite de l'attente avant de donner un résultat et qui supporte ces nouvelles fonctionnalités Python, vous pouvez l'utiliser comme tel :

```
burgers = await get_burgers(2)
```



Le mot-clé important ici est `await`. Il informe Python qu'il faut attendre  que `get_burgers(2)` finisse d'effectuer ses opérations  avant de stocker les résultats dans la variable `burgers`. Grâce à cela, Python saura qu'il peut aller effectuer d'autres opérations   pendant ce temps (comme par exemple recevoir une autre requête).

Pour que `await` fonctionne, il doit être placé dans une fonction qui supporte l'asynchronicité. Pour que ça soit le cas, il faut déclarer cette dernière avec `async def` :

```
async def get_burgers(number: int):
    # Opérations asynchrones pour créer les burgers
    return burgers
```

...et non `def` :

```
# Ceci n'est pas asynchrone
def get_sequential_burgers(number: int):
    # Opérations asynchrones pour créer les burgers
    return burgers
```

Avec `async def`, Python sait que dans cette fonction il doit prendre en compte les expressions `await`, et qu'il peut mettre en pause  l'exécution de la fonction pour aller faire autre chose  avant de revenir.

Pour appeler une fonction définie avec `async def`, vous devez utiliser `await`. Donc ceci ne marche pas :

```
# Ceci ne fonctionne pas, car get_burgers a été défini avec async def
burgers = get_burgers(2)
```

Donc, si vous utilisez une bibliothèque qui nécessite que ses fonctions soient appelées avec `await`, vous devez définir la *fonction de chemin* en utilisant `async def` comme dans :

```
@app.get('/burgers')
async def read_burgers():
    burgers = await get_burgers(2)
    return burgers
```

Plus de détails techniques

Vous avez donc compris que `await` peut seulement être utilisé dans des fonctions définies avec `async def`.

Mais en même temps, les fonctions définies avec `async def` doivent être appelées avec `await` et donc dans des fonctions définies elles aussi avec `async def`.

Vous avez donc remarqué ce paradoxe d'oeuf et de la poule, comment appelle-t-on la première fonction `async` ?

Si vous utilisez **FastAPI**, pas besoin de vous en inquiéter, car cette "première" fonction sera votre *fonction de chemin* ; et **FastAPI** saura comment arriver au résultat attendu.

Mais si vous utilisez `async` / `await` sans **FastAPI**, [allez jetez un coup d'oeil à la documentation officielle de Python](#).

Autres formes de code asynchrone

L'utilisation d' `async` et `await` est relativement nouvelle dans ce langage.

Mais cela rend la programmation asynchrone bien plus simple.


Cette même syntaxe (ou presque) était aussi incluse dans les versions modernes de Javascript (dans les versions navigateur et NodeJS).

Mais avant ça, gérer du code asynchrone était bien plus complexe et difficile.

Dans les versions précédentes de Python, vous auriez utilisé des *threads* ou [Gevent](#). Mais le code aurait été bien plus difficile à comprendre, déboguer, et concevoir.

Dans les versions précédentes de Javascript NodeJS / Navigateur, vous auriez utilisé des "callbacks". Menant potentiellement à ce que l'on appelle [le "callback hell"](#).

Coroutines

Coroutine est juste un terme élaboré pour désigner ce qui est retourné par une fonction définie avec `async def`. Python sait que c'est comme une fonction classique qui va démarrer à un moment et terminer à un autre, mais qu'elle peut aussi être mise en pause , du moment qu'il y a un `await` dans son contenu.

Mais toutes ces fonctionnalités d'utilisation de code asynchrone avec `async` et `await` sont souvent résumées comme l'utilisation des *coroutines*. On peut comparer cela à la principale fonctionnalité clé de Go, les "Goroutines".

Conclusion

Reprenons la phrase du début de la page :

Les versions modernes de Python supportent le **code asynchrone** grâce aux "**coroutines**" avec les syntaxes `async` et `await`.

Ceci devrait être plus compréhensible désormais. ✨

Tout ceci est donc ce qui donne sa force à **FastAPI** (à travers Starlette) et lui permet d'avoir des performances aussi impressionnantes.

Détails très techniques

!!! warning "Attention !" Vous pouvez probablement ignorer cela.

Ce sont des détails très poussés sur comment **FastAPI** fonctionne en arrière-plan.

Si vous avez de bonnes connaissances techniques (coroutines, threads, code bloquant, etc.) et êtes curieux de comment **FastAPI** gère `async def` versus le `def` classique, cette partie est faite pour vous.

Fonctions de chemin

Quand vous déclarez une *fonction de chemin* avec un `def` normal et non `async def`, elle est exécutée dans un groupe de threads (threadpool) externe qui est ensuite attendu, plutôt que d'être appelée directement (car cela bloquerait le serveur).

Si vous venez d'un autre framework asynchrone qui ne fonctionne pas comme de la façon décrite ci-dessus et que vous êtes habitués à définir des *fonctions de chemin* basiques avec un simple `def` pour un faible gain de performance (environ 100 nanosecondes), veuillez noter que dans **FastAPI**, l'effet serait plutôt contraire. Dans ces cas-là, il vaut mieux utiliser `async def` à moins que votre *fonction de chemin* utilise du code qui effectue des opérations I/O bloquantes.

Au final, dans les deux situations, il est fort probable que **FastAPI** soit tout de même [plus rapide](#)`{internal-link target=_blank}` que (ou au moins de vitesse égale à) votre framework précédent.

Dépendances

La même chose s'applique aux dépendances. Si une dépendance est définie avec `def` plutôt que `async def`, elle est exécutée dans la threadpool externe.

Sous-dépendances

Vous pouvez avoir de multiples dépendances et sous-dépendances dépendant les unes des autres (en tant que paramètres de la définition de la *fonction de chemin*), certaines créées avec `async def` et d'autres avec `def`. Cela fonctionnerait aussi, et celles définies avec un simple `def` seraient exécutées sur un thread externe (venant de la threadpool) plutôt que d'être "attendues".

Autres fonctions utilitaires

Toute autre fonction utilitaire que vous appelez directement peut être créée avec un classique `def` ou avec `async def` et **FastAPI** n'aura pas d'impact sur la façon dont vous l'appelez.

Contrairement aux fonctions que **FastAPI** appelle pour vous : les *fonctions de chemin* et dépendances.

Si votre fonction utilitaire est une fonction classique définie avec `def`, elle sera appelée directement (telle qu'écrite dans votre code), pas dans une threadpool, si la fonction est définie avec `async def` alors vous devrez attendre (avec `await`) que cette fonction se termine avant de passer à la suite du code.

Encore une fois, ce sont des détails très techniques qui peuvent être utiles si vous venez ici les chercher.

Sinon, les instructions de la section [Vous êtes pressés ?](#) ci-dessus sont largement suffisantes.