

# Concurrencia y async / await

Detalles sobre la sintaxis `async def` para *path operation functions* y un poco de información sobre código asíncrono, concurrencia y paralelismo.

## ¿Tienes prisa?

### TL:DR:

Si estás utilizando libraries de terceros que te dicen que las llares con `await`, del tipo:

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

Entonces declara tus *path operation functions* con `async def` de la siguiente manera:

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

!!! note "Nota" Solo puedes usar `await` dentro de funciones creadas con `async def`.

---

Si estás utilizando libraries de terceros que se comunican con algo (una base de datos, una API, el sistema de archivos, etc.) y no tienes soporte para `await` (este es el caso para la mayoría de las libraries de bases de datos), declara tus *path operation functions* de forma habitual, con solo `def`, de la siguiente manera:

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

---

Si tu aplicación (de alguna manera) no tiene que comunicarse con nada más y en consecuencia esperar a que responda, usa `def` normal.

---

Si simplemente no lo sabes, usa `def` normal.

---

**Nota:** puedes mezclar `def` y `async def` en tus *path operation functions* tanto como lo necesites y definir cada una utilizando la mejor opción para ti. FastAPI hará lo correcto con ellos.

De todos modos, en cualquiera de los casos anteriores, FastAPI seguirá funcionando de forma asíncrona y será extremadamente rápido.

Pero siguiendo los pasos anteriores, FastAPI podrá hacer algunas optimizaciones de rendimiento.

## Detalles Técnicos

Las versiones modernas de Python tienen soporte para "código asíncrono" usando algo llamado "coroutines", usando la sintaxis `async` y `await`.

Veamos esa frase por partes en las secciones siguientes:

- **Código Asíncrono**
- `async y await`
- **Coroutines**

## Código Asíncrono

El código asíncrono sólo significa que el lenguaje 🗨️ tiene una manera de decirle al sistema / programa 🤖 que, en algún momento del código, 🤖 tendrá que esperar a que *algo más* termine en otro sitio. Digamos que ese *algo más* se llama, por ejemplo, "archivo lento" 📄.

Durante ese tiempo, el sistema puede hacer otras cosas, mientras "archivo lento" 📄 termina.

Entonces el sistema / programa 🤖 volverá cada vez que pueda, sea porque está esperando otra vez, porque 🤖 ha terminado todo el trabajo que tenía en ese momento. Y 🤖 verá si alguna de las tareas por las que estaba esperando ha terminado, haciendo lo que tenía que hacer.

Luego, 🤖 cogerá la primera tarea finalizada (digamos, nuestro "archivo lento" 📄) y continuará con lo que tenía que hacer con esa tarea.

Esa "espera de otra cosa" normalmente se refiere a operaciones I/O que son relativamente "lentas" (en relación a la velocidad del procesador y memoria RAM), como por ejemplo esperar por:

- los datos de cliente que se envían a través de la red
- los datos enviados por tu programa para ser recibidos por el cliente a través de la red
- el contenido de un archivo en disco para ser leído por el sistema y entregado al programa
- los contenidos que tu programa da al sistema para ser escritos en disco
- una operación relacionada con una API remota
- una operación de base de datos
- el retorno de resultados de una consulta de base de datos
- etc.

Como el tiempo de ejecución se consume principalmente al esperar a operaciones de I/O, las llaman operaciones "I/O bound".

Se llama "asíncrono" porque el sistema / programa no tiene que estar "sincronizado" con la tarea lenta, esperando el momento exacto en que finaliza la tarea, sin hacer nada, para poder recoger el resultado de la tarea y continuar el trabajo.

En lugar de eso, al ser un sistema "asíncrono", una vez finalizada, la tarea puede esperar un poco en la cola (algunos microsegundos) para que la computadora / programa termine lo que estaba haciendo, y luego vuelva para recoger los resultados y seguir trabajando con ellos.

Por "síncrono" (contrario a "asíncrono") también se usa habitualmente el término "secuencial", porque el sistema / programa sigue todos los pasos secuencialmente antes de cambiar a una tarea diferente, incluso si esos pasos implican esperas.

## Concurrencia y Hamburguesas

El concepto de código **asíncrono** descrito anteriormente a veces también se llama "**concurrencia**". Es diferente del "**paralelismo**".

**Concurrencia** y **paralelismo** ambos se relacionan con "cosas diferentes que suceden más o menos al mismo tiempo".

Pero los detalles entre *concurrentencia* y *paralelismo* son bastante diferentes.

Para entender las diferencias, imagina la siguiente historia sobre hamburguesas:

## Hamburguesas Concurrentes

Vas con la persona que te gusta 🧑🏻 a pedir comida rápida 🍔, haces cola mientras el cajero 🧑🏻 recoge los pedidos de las personas de delante tuyo.

Llega tu turno, haces tu pedido de 2 hamburguesas impresionantes para esa persona 🧑🏻 y para ti.

Pagas 💵.

El cajero 🧑🏻 le dice algo al chico de la cocina 🧑🏻 para que sepa que tiene que preparar tus hamburguesas 🍔 (a pesar de que actualmente está preparando las de los clientes anteriores).

El cajero 🧑🏻 te da el número de tu turno.

Mientras esperas, vas con esa persona 🧑🏻 y eliges una mesa, se sientan y hablan durante un rato largo (ya que las hamburguesas son muy impresionantes y necesitan un rato para prepararse 🌟🍔🌟).

Mientras te sientas en la mesa con esa persona 🧑🏻, esperando las hamburguesas 🍔, puedes disfrutar ese tiempo admirando lo increíble, inteligente, y bien que se ve 🌟🧑🏻🌟.

Mientras esperas y hablas con esa persona 🧑🏻, de vez en cuando, verificas el número del mostrador para ver si ya es tu turno.

Al final, en algún momento, llega tu turno. Vas al mostrador, coges tus hamburguesas 🍔 y vuelves a la mesa.

Tú y esa persona 🧑🏻 se comen las hamburguesas 🍔 y la pasan genial 🌟.

---

Imagina que eres el sistema / programa 🤖 en esa historia.

Mientras estás en la cola, estás quieto 🧘, esperando tu turno, sin hacer nada muy "productivo". Pero la línea va rápida porque el cajero 🧑🏻 solo recibe los pedidos (no los prepara), así que está bien.

Luego, cuando llega tu turno, haces un trabajo "productivo" real 🧑🏻, procesas el menú, decides lo que quieres, lo que quiere esa persona 🧑🏻, pagas 💵, verificas que das el billete o tarjeta correctos, verificas que te cobren correctamente, que el pedido tiene los artículos correctos, etc.

Pero entonces, aunque aún no tienes tus hamburguesas 🍔, el trabajo hecho con el cajero 🧑🏻 está "en pausa" ⏸, porque debes esperar ⌚ a que tus hamburguesas estén listas.

Pero como te alejas del mostrador y te sientas en la mesa con un número para tu turno, puedes cambiar tu atención 🔄 a esa persona 🧑🏻 y "trabajar" 🏠🧑🏻 en eso. Entonces nuevamente estás haciendo algo muy "productivo" 🧑🏻, como coquetear con esa persona 🧑🏻.

Después, el 🧑🏻 cajero dice "he terminado de hacer las hamburguesas" 🍔 poniendo tu número en la pantalla del mostrador, pero no saltas al momento que el número que se muestra es el tuyo. Sabes que nadie robará tus hamburguesas 🍔 porque tienes el número de tu turno y ellos tienen el suyo.

Así que esperas a que esa persona 🧑🏻 termine la historia (terminas el trabajo actual 🏠 / tarea actual que se está procesando 🧑🏻), sonríes gentilmente y le dices que vas por las hamburguesas ⏸.

Luego vas al mostrador 🔄, a la tarea inicial que ya está terminada 🏠, recoges las hamburguesas 🍔, les dices gracias y las llevas a la mesa. Eso termina esa fase / tarea de interacción con el mostrador 🏠. Eso a su vez, crea una nueva tarea, "comer hamburguesas" 🔄🏠, pero la anterior de "conseguir hamburguesas" está terminada 🏠.

## Hamburguesas Paralelas

Ahora imagina que estas no son "Hamburguesas Concurrentes" sino "Hamburguesas Paralelas".

Vas con la persona que te gusta 🥰 por comida rápida paralela 🍔.

Haces la cola mientras varios cajeros (digamos 8) que a la vez son cocineros 🍳🍳🍳🍳🍳🍳🍳🍳 toman los pedidos de las personas que están delante de ti.

Todos los que están antes de ti están esperando ⌚ que sus hamburguesas 🍔 estén listas antes de dejar el mostrador porque cada uno de los 8 cajeros prepara la hamburguesa de inmediato antes de recibir el siguiente pedido.

Entonces finalmente es tu turno, haces tu pedido de 2 hamburguesas 🍔 impresionantes para esa persona 🥰 y para ti.

Pagas 💵.

El cajero va a la cocina 🍳.

Esperas, de pie frente al mostrador ⌚, para que nadie más recoja tus hamburguesas 🍔, ya que no hay números para los turnos.

Como tu y esa persona 🥰 están ocupados en impedir que alguien se ponga delante y recoja tus hamburguesas apenas llegan ⌚, tampoco puedes prestarle atención a esa persona 😞.

Este es un trabajo "síncrono", estás "sincronizado" con el cajero / cocinero 🍳. Tienes que esperar y estar allí en el momento exacto en que el cajero / cocinero 🍳 termina las hamburguesas 🍔 y te las da, o de lo contrario, alguien más podría cogerlas.

Luego, el cajero / cocinero 🍳 finalmente regresa con tus hamburguesas 🍔, después de mucho tiempo esperando ⌚ frente al mostrador.

Cojes tus hamburguesas 🍔 y vas a la mesa con esa persona 🥰.

Sólo las comes y listo 🍔 🍷.

No has hablado ni coqueteado mucho, ya que has pasado la mayor parte del tiempo esperando ⌚ frente al mostrador 😞.

---

En este escenario de las hamburguesas paralelas, tú eres un sistema / programa 🖥 con dos procesadores (tú y la persona que te gusta 🥰), ambos esperando ⌚ y dedicando su atención 🎮 a estar "esperando en el mostrador" ⌚ durante mucho tiempo.

La tienda de comida rápida tiene 8 procesadores (cajeros / cocineros) 🍳🍳🍳🍳🍳🍳🍳🍳. Mientras que la tienda de hamburguesas concurrentes podría haber tenido solo 2 (un cajero y un cocinero) 🍳🍳.

Pero aún así, la experiencia final no es la mejor 😞.

---

Esta sería la historia paralela equivalente de las hamburguesas 🍔.

Para un ejemplo más "real" de ésto, imagina un banco.

Hasta hace poco, la mayoría de los bancos tenían varios cajeros 🧑🏦🧑🏦🧑🏦 y una gran línea ⌚⌚⌚⌚⌚⌚⌚⌚.

Todos los cajeros haciendo todo el trabajo con un cliente tras otro 🧑🏦 🎮.

Y tienes que esperar 🕒 en la fila durante mucho tiempo o perderás tu turno.

Probablemente no querrás llevar contigo a la persona que te gusta 😍 a hacer encargos al banco 🏦.

## Conclusión de las Hamburguesa

En este escenario de "hamburguesas de comida rápida con tu pareja", debido a que hay mucha espera 🕒, tiene mucho más sentido tener un sistema con concurrencia 🔄.

Este es el caso de la mayoría de las aplicaciones web.

Muchos, muchos usuarios, pero el servidor está esperando 🕒 el envío de las peticiones ya que su conexión no es buena.

Y luego esperando 🕒 nuevamente a que las respuestas retornen.

Esta "espera" 🕒 se mide en microsegundos, pero aun así, sumando todo, al final es mucha espera.

Es por eso que tiene mucho sentido usar código asíncrono 🔄 para las API web.

La mayoría de los framework populares de Python existentes (incluidos Flask y Django) se crearon antes de que existieran las nuevas funciones asíncronas en Python. Por lo tanto, las formas en que pueden implementarse admiten la ejecución paralela y una forma más antigua de ejecución asíncrona que no es tan potente como la actual.

A pesar de que la especificación principal para Python web asíncrono (ASGI) se desarrolló en Django, para agregar soporte para WebSockets.

Ese tipo de asincronía es lo que hizo popular a NodeJS (aunque NodeJS no es paralelo) y esa es la fortaleza de Go como lenguaje de programación.

Y ese es el mismo nivel de rendimiento que obtienes con **FastAPI**.

Y como puede tener paralelismo y asincronía al mismo tiempo, obtienes un mayor rendimiento que la mayoría de los frameworks de NodeJS probados y a la par con Go, que es un lenguaje compilado más cercano a C ([todo gracias Starlette](#)).

## ¿Es la concurrencia mejor que el paralelismo?

¡No! Esa no es la moraleja de la historia.

La concurrencia es diferente al paralelismo. Y es mejor en escenarios **específicos** que implican mucha espera. Debido a eso, generalmente es mucho mejor que el paralelismo para el desarrollo de aplicaciones web. Pero no para todo.

Entonces, para explicar eso, imagina la siguiente historia corta:

*Tienes que limpiar una casa grande y sucia.*

*Sí, esa es toda la historia.*

---

No hay esperas 🕒, solo hay mucho trabajo por hacer, en varios lugares de la casa.

Podrías tener turnos como en el ejemplo de las hamburguesas, primero la sala de estar, luego la cocina, pero como no estás esperando nada, solo limpiando y limpiando, los turnos no afectarían nada.

Tomaría la misma cantidad de tiempo terminar con o sin turnos (concurrencia) y habrías hecho la misma cantidad de trabajo.

Pero en este caso, si pudieras traer a los 8 ex cajeros / cocineros / ahora limpiadores 🧑🧑🧑🧑🧑🧑🧑🧑, y cada uno de ellos (y tú) podría tomar una zona de la casa para limpiarla, podría hacer todo el trabajo en **paralelo**, con la ayuda adicional y terminar mucho antes.

En este escenario, cada uno de los limpiadores (incluido tú) sería un procesador, haciendo su parte del trabajo.

Y como la mayor parte del tiempo de ejecución lo coge el trabajo real (en lugar de esperar), y el trabajo en un sistema lo realiza una CPU, a estos problemas se les llama "CPU bond".

---

Ejemplos típicos de operaciones dependientes de CPU son cosas que requieren un procesamiento matemático complejo.

Por ejemplo:

- **Audio o procesamiento de imágenes.**
- **Visión por computadora:** una imagen está compuesta de millones de píxeles, cada píxel tiene 3 valores / colores, procesamiento que normalmente requiere calcular algo en esos píxeles, todo al mismo tiempo.
- **Machine Learning:** normalmente requiere muchas multiplicaciones de "matrices" y "vectores". Imagina en una enorme hoja de cálculo con números y tener que multiplicarlos todos al mismo tiempo.
- **Deep Learning:** este es un subcampo de Machine Learning, por lo tanto, aplica lo mismo. Es solo que no hay una sola hoja de cálculo de números para multiplicar, sino un gran conjunto de ellas, y en muchos casos, usa un procesador especial para construir y / o usar esos modelos.

## Concurrencia + Paralelismo: Web + Machine Learning

Con **FastAPI** puedes aprovechar la concurrencia que es muy común para el desarrollo web (atractivo principal de NodeJS).

Pero también puedes aprovechar los beneficios del paralelismo y el multiprocesamiento (tener múltiples procesos ejecutándose en paralelo) para cargas de trabajo **CPU bond** como las de los sistemas de Machine Learning.

Eso, más el simple hecho de que Python es el lenguaje principal para **Data Science**, Machine Learning y especialmente Deep Learning, hacen de FastAPI una muy buena combinación para las API y aplicaciones web de Data Science / Machine Learning (entre muchas otras).





Para ver cómo lograr este paralelismo en producción, consulta la sección sobre [Despliegue](#) (target=\_blank).

### async y await

Las versiones modernas de Python tienen una forma muy intuitiva de definir código asíncrono. Esto hace que se vea como un código "secuencial" normal y que haga la "espera" por ti en los momentos correctos.

Cuando hay una operación que requerirá esperar antes de dar los resultados y tiene soporte para estas nuevas características de Python, puedes programarlo como:

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



La clave aquí es `await`. Eso le dice a Python que tiene que esperar  a que `get_burgers(2)` termine de hacer lo suyo  antes de almacenar los resultados en `hamburguesas`. Con eso, Python sabrá que puede ir y hacer otra cosa   mientras tanto (como recibir otra solicitud).

Para que `await` funcione, tiene que estar dentro de una función que admita esta asincronía. Para hacer eso, simplemente lo declaras con `async def`:

```
async def get_burgers(number: int):
    # Do some asynchronous stuff to create the burgers
    return burgers
```

...en vez de `def` :

```
# This is not asynchronous
def get_sequential_burgers(number: int):
    # Do some sequential stuff to create the burgers
    return burgers
```

Con `async def` , Python sabe que, dentro de esa función, debe tener en cuenta las expresiones `await` y que puede "pausar"  la ejecución de esa función e ir a hacer otra cosa  antes de regresar.

Cuando desees llamar a una función `async def` , debes "esperarla". Entonces, esto no funcionará:

```
# Esto no funcionará, porque get_burgers se definió con: async def
hamburguesas = get_burgers (2)
```

---

Por lo tanto, si estás utilizando una library que te dice que puedes llamarla con `await` , debes crear las *path operation functions* que la usan con `async def` , como en:

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

## Más detalles técnicos

Es posible que hayas notado que `await` solo se puede usar dentro de las funciones definidas con `async def` .

Pero al mismo tiempo, las funciones definidas con `async def` deben ser "esperadas". Por lo tanto, las funciones con `async def` solo se pueden invocar dentro de las funciones definidas con `async def` también.

Entonces, relacionado con la paradoja del huevo y la gallina, ¿cómo se llama a la primera función `async` ?

Si estás trabajando con **FastAPI** no tienes que preocuparte por eso, porque esa "primera" función será tu *path operation function*, y FastAPI sabrá cómo hacer lo pertinente.

En el caso de que desees usar `async` / `await` sin FastAPI, [revisa la documentación oficial de Python](#).

## Otras formas de código asíncrono

Este estilo de usar `async` y `await` es relativamente nuevo en el lenguaje.

Pero hace que trabajar con código asíncrono sea mucho más fácil.


Esta misma sintaxis (o casi idéntica) también se incluyó recientemente en las versiones modernas de JavaScript (en Browser y NodeJS).

Pero antes de eso, manejar código asíncrono era bastante más complejo y difícil.

En versiones anteriores de Python, podrías haber utilizado `threads` o [Gevent](#). Pero el código es mucho más complejo de entender, depurar y desarrollar.

En versiones anteriores de NodeJS / Browser JavaScript, habrías utilizado "callbacks". Lo que conduce a [callback hell](#).

## Coroutines

**Coroutine** es un término sofisticado para referirse a la cosa devuelta por una función `async def`. Python sabe que es algo así como una función que puede iniciar y que terminará en algún momento, pero que también podría pausarse  internamente, siempre que haya un `await` dentro de ella.

Pero toda esta funcionalidad de usar código asíncrono con `async` y `await` se resume muchas veces como usar "coroutines". Es comparable a la característica principal de Go, las "Goroutines".

## Conclusión

Veamos la misma frase de arriba:

*Las versiones modernas de Python tienen soporte para "código asíncrono" usando algo llamado "coroutines", con la sintaxis `async` y `await`.*

Eso ya debería tener más sentido ahora. ✨

Todo eso es lo que impulsa FastAPI (a través de Starlette) y lo que hace que tenga un rendimiento tan impresionante.

## Detalles muy técnicos

!!! warning "Advertencia" Probablemente puedas saltarte esto.

Estos son detalles muy técnicos de cómo **FastAPI** funciona a muy bajo nivel.

Si tienes bastante conocimiento técnico (coroutines, threads, bloqueos, etc.) y tienes curiosidad acerca de cómo FastAPI gestiona `async def` vs `def` normal, continúa.

### Path operation functions

Cuando declaras una *path operation function* con `def` normal en lugar de `async def`, se ejecuta en un threadpool externo que luego es "awaited", en lugar de ser llamado directamente (ya que bloquearía el servidor).

Si vienes de otro framework asíncrono que no funciona de la manera descrita anteriormente y estás acostumbrado a definir *path operation functions* del tipo sólo cálculo con `def` simple para una pequeña ganancia de rendimiento (aproximadamente 100 nanosegundos), ten en cuenta que en **FastAPI** el efecto sería bastante opuesto. En estos casos, es mejor usar `async def` a menos que tus *path operation functions* usen un código que realice el bloqueo I/O.

Aún así, en ambas situaciones, es probable que **FastAPI** sea [aún más rápido](#) (Internal-link target=\_blank) que (o al menos comparable) a tu framework anterior.

### Dependencias

Lo mismo se aplica para las dependencias. Si una dependencia es una función estándar `def` en lugar de `async def`, se ejecuta en el threadpool externo.



## Subdependencias

Puedes tener múltiples dependencias y subdependencias que se requieren unas a otras (como parámetros de las definiciones de cada función), algunas de ellas pueden crearse con `async def` y otras con `def` normal. Igual todo seguiría funcionando correctamente, y las creadas con `def` normal se llamarían en un thread externo (del threadpool) en lugar de ser "awaited".

## Otras funciones de utilidades

Cualquier otra función de utilidad que llames directamente se puede crear con `def` o `async def` normales y FastAPI no afectará la manera en que la llames.

Esto contrasta con las funciones que FastAPI llama por ti: las *path operation functions* y dependencias.

Si tu función de utilidad es creada con `def` normal, se llamará directamente (tal cual la escribes en tu código), no en un threadpool, si la función se crea con `async def`, entonces debes usar `await` con esa función cuando la llamas en tu código.

---

Nuevamente, estos son detalles muy técnicos que probablemente sólo son útiles si los viniste a buscar expresamente.

De lo contrario, la guía de la sección anterior debería ser suficiente: [¿Tienes prisa?](#).