FastAPI Learn

Concurrency and async / await

Details about the `async def` syntax for *path operation functions* and some background about asynchronous code, concurrency, and parallelism.

In a hurry?

TL;DR:

If you are using third party libraries that tell you to call them with `await`, like:

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

Then, declare your *path operation functions* with `async def` like:

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

Note

You can only use `await` inside of functions created with `async def`.

If you are using a third party library that communicates with something (a database, an API, the file system, etc.) and doesn't have support for using `await`, (this is currently the case for most database libraries), then declare your *path operation functions* as normally, with just `def`, like:

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

If your application (somehow) doesn't have to communicate with anything else and wait for it to respond, use `async def`, even if you don't need to use `await` inside.

If you just don't know, use normal `def`.

Note: You can mix `def` and `async def` in your *path operation functions* as much as you need and define each one using the best option for you. FastAPI will do the right thing with them.

Anyway, in any of the cases above, FastAPI will still work asynchronously and be extremely fast.

But by following the steps above, it will be able to do some performance optimizations.

Technical Details

Modern versions of Python have support for **"asynchronous code"** using something called **"coroutines"**, with **`async` and `await`** syntax.

Let's see that phrase by parts in the sections below:

- **Asynchronous Code**
- **`async` and `await`**
- **Coroutines**

Asynchronous Code

Asynchronous code just means that the language 💬 has a way to tell the computer / program 🤖 that at some point in the code, it 🤖 will have to wait for *something else* to finish somewhere else. Let's say that *something else* is called "slow-file" 📝.

So, during that time, the computer can go and do some other work, while "slow-file" 📝 finishes.

Then the computer / program 🤖 will come back every time it has a chance because it's waiting again, or whenever it 🤖 finished all the work it had at that point. And it 🤖 will see if any of the tasks it was waiting for have already finished, doing whatever it had to do.

Next, it 🤖 takes the first task to finish (let's say, our "slow-file" 📝) and continues whatever it had to do with it.

That "wait for something else" normally refers to I/O operations that are relatively "slow" (compared to the speed of the processor and the RAM memory), like waiting for:

- the data from the client to be sent through the network
- the data sent by your program to be received by the client through the network
- the contents of a file in the disk to be read by the system and given to your program
- the contents your program gave to the system to be written to disk
- a remote API operation
- a database operation to finish
- a database query to return the results
- etc.

As the execution time is consumed mostly by waiting for I/O operations, they call them "I/O bound" operations.

It's called "asynchronous" because the computer / program doesn't have to be "synchronized" with the slow task, waiting for the exact moment that the task finishes, while doing nothing, to be able to take the task result and continue the work.

Instead of that, by being an "asynchronous" system, once finished, the task can wait in line a little bit (some microseconds) for the computer / program to finish whatever it went to do, and then come back to take the results and continue working with them.

For "synchronous" (contrary to "asynchronous") they commonly also use the term "sequential", because the computer / program follows all the steps in sequence before switching to a different task, even if those steps involve waiting.

Concurrency and Burgers

This idea of **asynchronous** code described above is also sometimes called **"concurrency"**. It is different from **"parallelism"**.

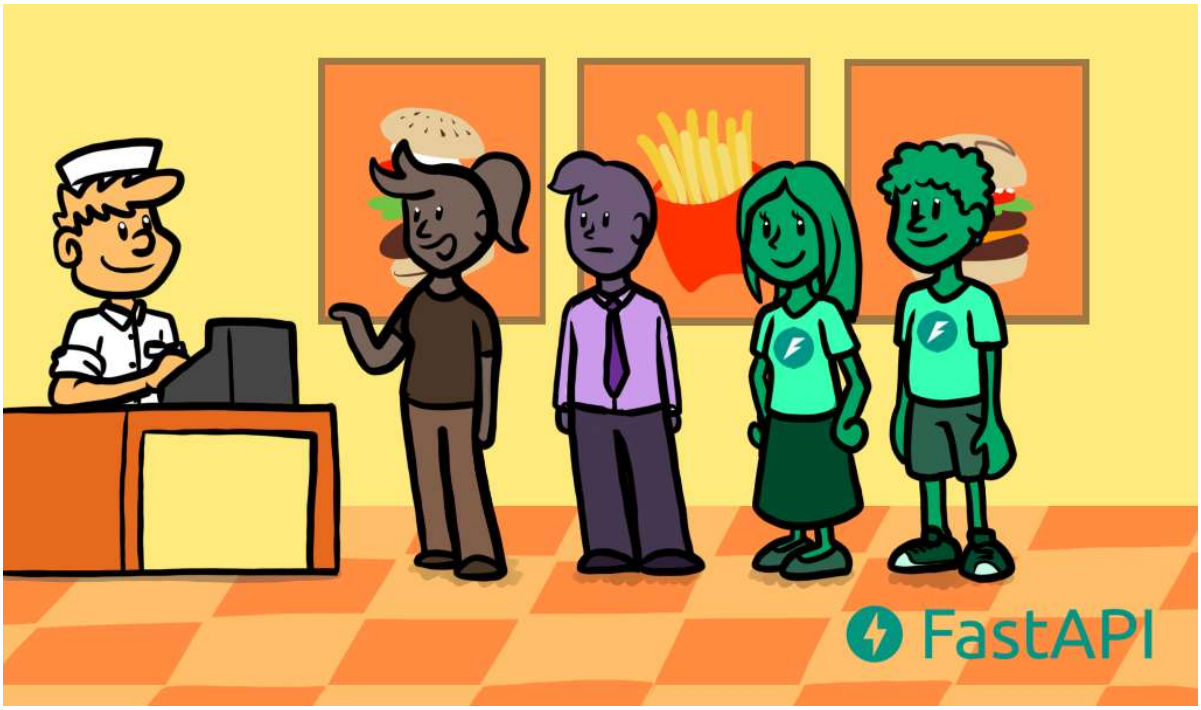
Concurrency and **parallelism** both relate to "different things happening more or less at the same time".

But the details between *concurrency* and *parallelism* are quite different.

To see the difference, imagine the following story about burgers:

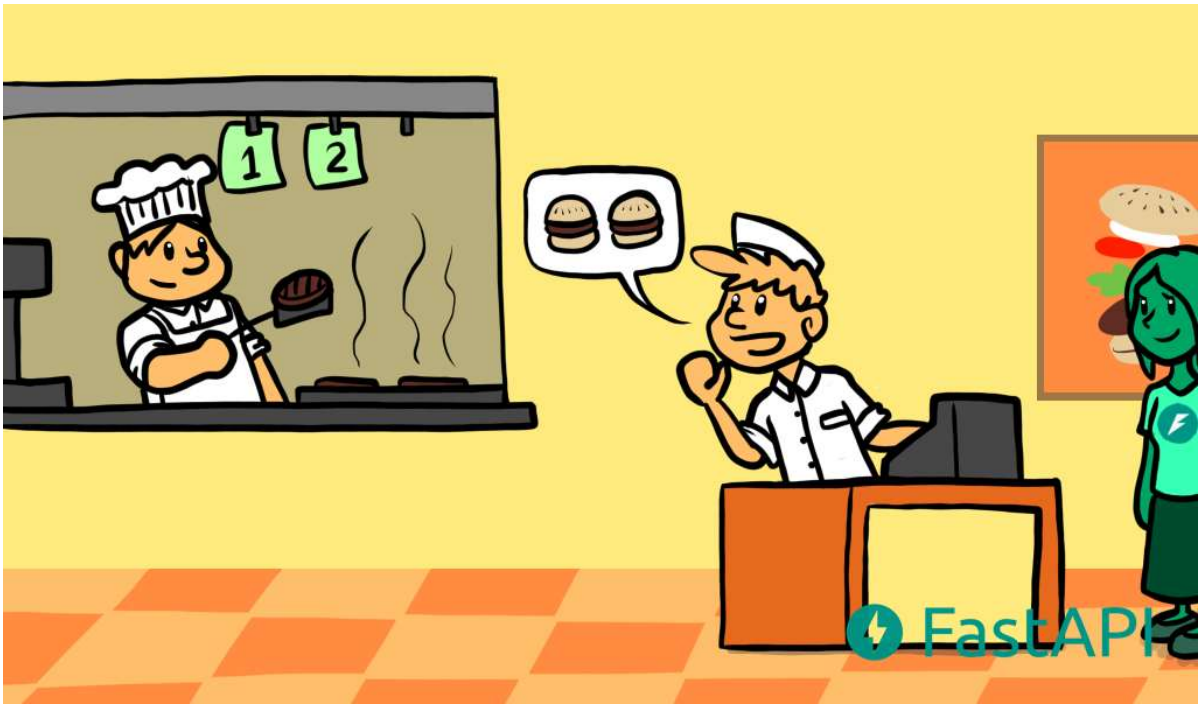
Concurrent Burgers

You go with your crush to get fast food, you stand in line while the cashier takes the orders from the people in front of you. 😍



Then it's your turn, you place your order of 2 very fancy burgers for your crush and you. 🍔🍔

The cashier says something to the cook in the kitchen so they know they have to prepare your burgers (even though they are currently preparing the ones for the previous clients).



You pay. 💸

The cashier gives you the number of your turn.

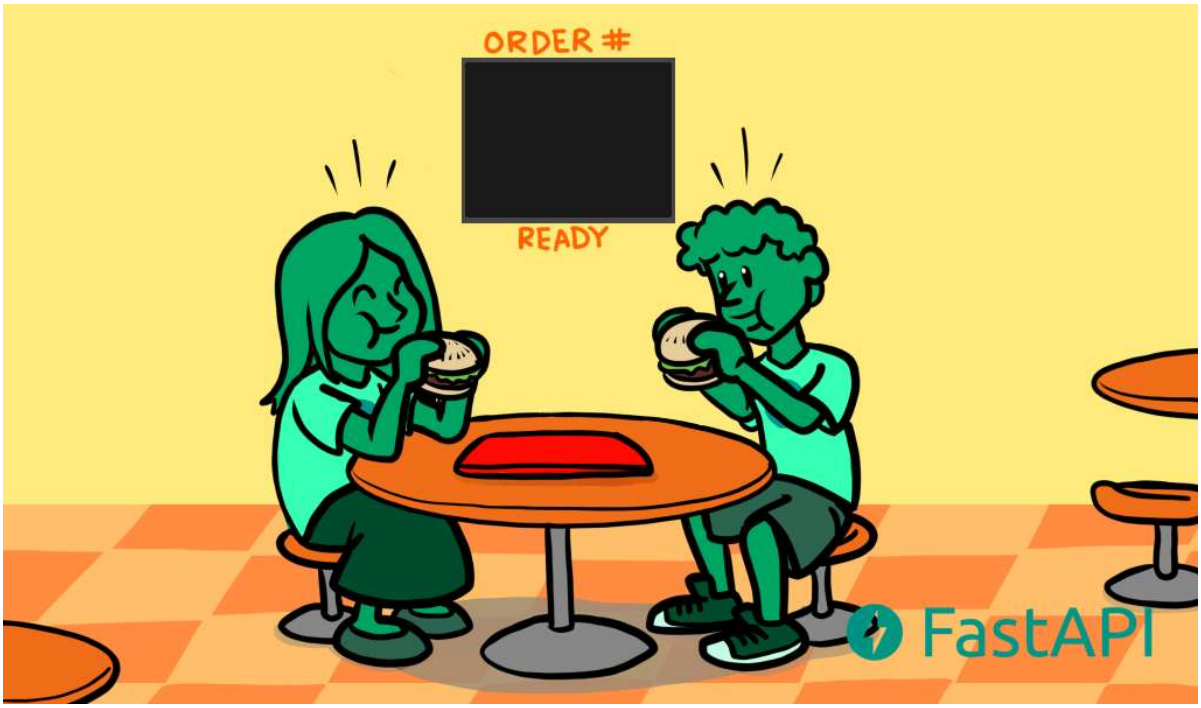
While you are waiting, you go with your crush and pick a table, you sit and talk with your crush for a long time (as your burgers are very fancy and take some time to prepare).

As you are sitting at the table with your crush, while you wait for the burgers, you can spend that time admiring how awesome, cute and smart your crush is ✨😍✨.

While waiting and talking to your crush, from time to time, you check the number displayed on the counter to see if it's your turn already.

Then at some point, it finally is your turn. You go to the counter, get your burgers and come back to the table.

You and your crush eat the burgers and have a nice time. ✨



Info

Beautiful illustrations by Ketrina Thompson [🔜]. 🎨

Imagine you are the computer / program 🤖 in that story.

While you are at the line, you are just idle 😴, waiting for your turn, not doing anything very "productive". But the line is fast because the cashier is only taking the orders (not preparing them), so that's fine.

Then, when it's your turn, you do actual "productive" work, you process the menu, decide what you want, get your crush's choice, pay, check that you give the correct bill or card, check that you are charged correctly, check that the order has the correct items, etc.

But then, even though you still don't have your burgers, your work with the cashier is "on pause" ⏸️, because you have to wait 🕙 for your burgers to be ready.

But as you go away from the counter and sit at the table with a number for your turn, you can switch 🔀 your attention to your crush, and "work" ⏯️ 🤓 on that. Then you are again doing

something very "productive" as is flirting with your crush 😍.

Then the cashier 💁 says "I'm finished with doing the burgers" by putting your number on the counter's display, but you don't jump like crazy immediately when the displayed number changes to your turn number. You know no one will steal your burgers because you have the number of your turn, and they have theirs.

So you wait for your crush to finish the story (finish the current work ⏯ / task being processed 🤓), smile gently and say that you are going for the burgers ⏸.

Then you go to the counter 🔀, to the initial task that is now finished ⏯, pick the burgers, say thanks and take them to the table. That finishes that step / task of interaction with the counter ⏹. That in turn, creates a new task, of "eating burgers" 🔀 ⏯, but the previous one of "getting burgers" is finished ⏹.

Parallel Burgers

Now let's imagine these aren't "Concurrent Burgers", but "Parallel Burgers".

You go with your crush to get parallel fast food.

You stand in line while several (let's say 8) cashiers that at the same time are cooks take the orders from the people in front of you.

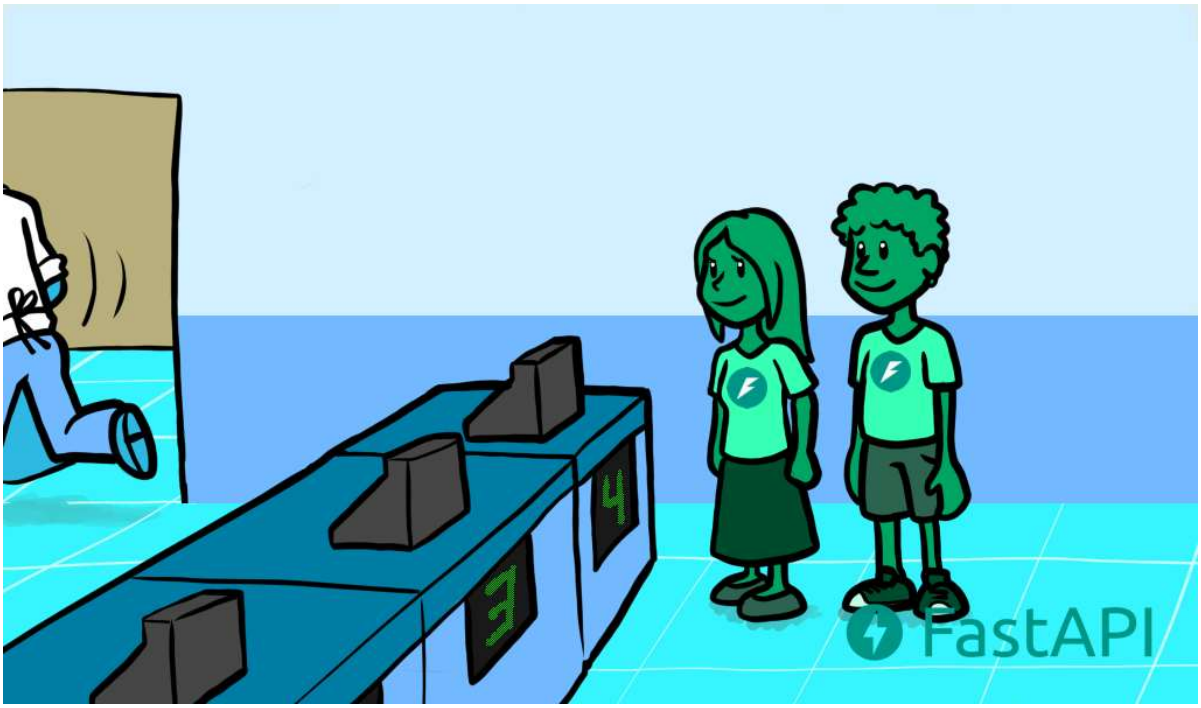
Everyone before you is waiting for their burgers to be ready before leaving the counter because each of the 8 cashiers goes and prepares the burger right away before getting the next order.

Then it's finally your turn, you place your order of 2 very fancy burgers for your crush and you.

You pay 💸.

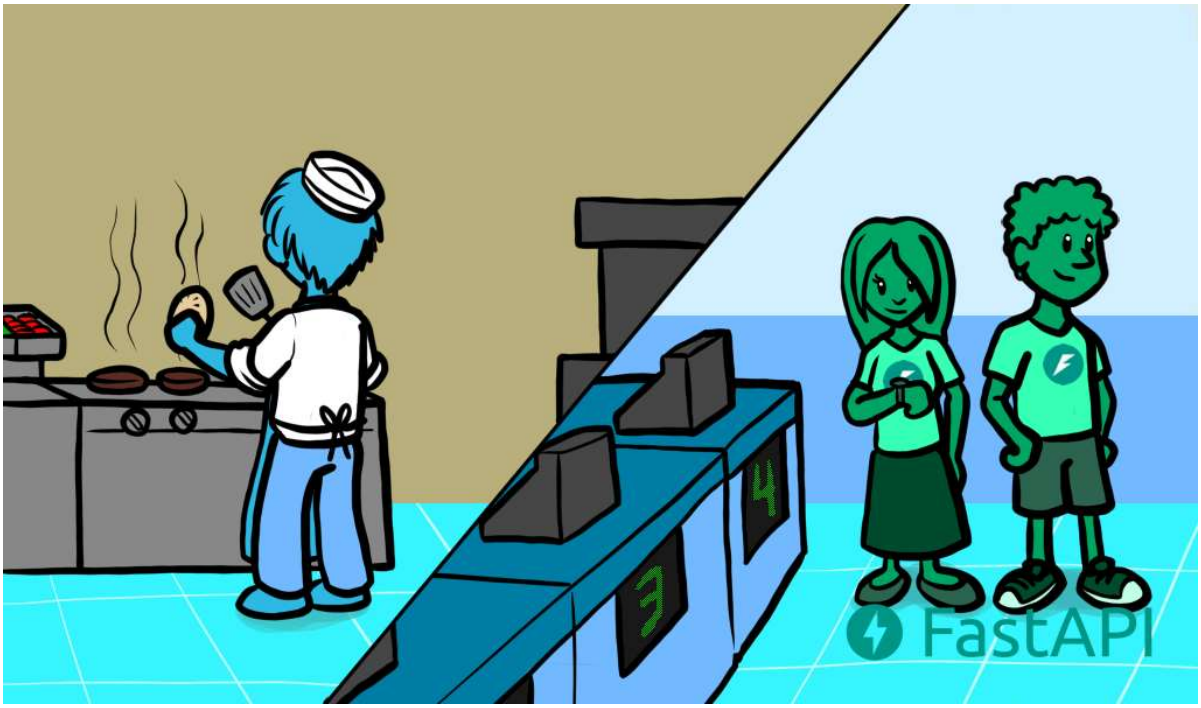
The cashier goes to the kitchen.

You wait, standing in front of the counter 🕙, so that no one else takes your burgers before you do, as there are no numbers for turns.



As you and your crush are busy not letting anyone get in front of you and take your burgers whenever they arrive, you cannot pay attention to your crush. 😞

This is "synchronous" work, you are "synchronized" with the cashier/cook 👨‍🍳. You have to wait 🕙 and be there at the exact moment that the cashier/cook 👨‍🍳 finishes the burgers and gives them to you, or otherwise, someone else might take them.



Then your cashier/cook 👨‍🍳 finally comes back with your burgers, after a long time waiting 🕙 there in front of the counter.

You take your burgers and go to the table with your crush.

You just eat them, and you are done. ⏹️

There was not much talk or flirting as most of the time was spent waiting 🕙 in front of the counter. 😞

> **Info**
>
> Beautiful illustrations by Ketrina Thompson [🔀]. 🎨

In this scenario of the parallel burgers, you are a computer / program 🤖 with two processors (you and your crush), both waiting 🕙 and dedicating their attention ⏯ to be "waiting on the counter" 🕙 for a long time.

The fast food store has 8 processors (cashiers/cooks). While the concurrent burgers store might have had only 2 (one cashier and one cook).

But still, the final experience is not the best. 😞

This would be the parallel equivalent story for burgers. 🍔

For a more "real life" example of this, imagine a bank.

Up to recently, most of the banks had multiple cashiers 👨‍💼👨‍💼👨‍💼👨‍💼 and a big line 🕙🕙🕙🕙🕙🕙🕙🕙.

All of the cashiers doing all the work with one client after the other 👨‍💼⏯.

And you have to wait 🕙 in the line for a long time or you lose your turn.

You probably wouldn't want to take your crush 😍 with you to run errands at the bank 🏦.

Burger Conclusion

In this scenario of "fast food burgers with your crush", as there is a lot of waiting 🕙, it makes a lot more sense to have a concurrent system ⏸🔀⏯.

This is the case for most of the web applications.

Many, many users, but your server is waiting 🕙 for their not-so-good connection to send their requests.

And then waiting 🕙 again for the responses to come back.

This "waiting" 🕙 is measured in microseconds, but still, summing it all, it's a lot of waiting in the end.

That's why it makes a lot of sense to use asynchronous ⏸🔀⏯ code for web APIs.

This kind of asynchronicity is what made NodeJS popular (even though NodeJS is not parallel) and that's the strength of Go as a programming language.

And that's the same level of performance you get with **FastAPI**.

And as you can have parallelism and asynchronicity at the same time, you get higher performance than most of the tested NodeJS frameworks and on par with Go, which is a compiled language closer to C (all thanks to Starlette) [🔁].

Is concurrency better than parallelism?

Nope! That's not the moral of the story.

Concurrency is different than parallelism. And it is better on **specific** scenarios that involve a lot of waiting. Because of that, it generally is a lot better than parallelism for web application development. But not for everything.

So, to balance that out, imagine the following short story:

> You have to clean a big, dirty house.

Yep, that's the whole story.

There's no waiting 🕙 anywhere, just a lot of work to be done, on multiple places of the house.

You could have turns as in the burgers example, first the living room, then the kitchen, but as you are not waiting 🕙 for anything, just cleaning and cleaning, the turns wouldn't affect anything.

It would take the same amount of time to finish with or without turns (concurrency) and you would have done the same amount of work.

But in this case, if you could bring the 8 ex-cashier/cooks/now-cleaners, and each one of them (plus you) could take a zone of the house to clean it, you could do all the work in **parallel**, with the extra help, and finish much sooner.

In this scenario, each one of the cleaners (including you) would be a processor, doing their part of the job.

And as most of the execution time is taken by actual work (instead of waiting), and the work in a computer is done by a CPU, they call these problems "CPU bound".

Common examples of CPU bound operations are things that require complex math processing.

For example:

- **Audio** or **image processing**.
- **Computer vision**: an image is composed of millions of pixels, each pixel has 3 values / colors, processing that normally requires computing something on those pixels, all at the same time.
- **Machine Learning**: it normally requires lots of "matrix" and "vector" multiplications. Think of a huge spreadsheet with numbers and multiplying all of them together at the same time.
- **Deep Learning**: this is a sub-field of Machine Learning, so, the same applies. It's just that there is not a single spreadsheet of numbers to multiply, but a huge set of them, and in many cases, you use a special processor to build and / or use those models.

Concurrency + Parallelism: Web + Machine Learning

With **FastAPI** you can take advantage of concurrency that is very common for web development (the same main attraction of NodeJS).

But you can also exploit the benefits of parallelism and multiprocessing (having multiple processes running in parallel) for **CPU bound** workloads like those in Machine Learning systems.

That, plus the simple fact that Python is the main language for **Data Science**, Machine Learning and especially Deep Learning, make FastAPI a very good match for Data Science / Machine Learning web APIs and applications (among many others).

To see how to achieve this parallelism in production see the section about Deployment ↪️.

`async` and `await`

Modern versions of Python have a very intuitive way to define asynchronous code. This makes it look just like normal "sequential" code and do the "awaiting" for you at the right moments.

When there is an operation that will require waiting before giving the results and has support for these new Python features, you can code it like:

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

The key here is the `await`. It tells Python that it has to wait ⏸ for `get_burgers(2)` to finish doing its thing 🕙 before storing the results in `burgers`. With that, Python will know that it can go and do something else 🔀 ⏯ in the meanwhile (like receiving another request).

For `await` to work, it has to be inside a function that supports this asynchronicity. To do that, you just declare it with `async def`:

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

...instead of `def`:

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

With `async def`, Python knows that, inside that function, it has to be aware of `await` expressions, and that it can "pause" ⏸ the execution of that function and go do something else 🔀 before coming back.

When you want to call an `async def` function, you have to "await" it. So, this won't work:

```
# This won't work, because get_burgers was defined with: async def
burgers = get_burgers(2)
```

So, if you are using a library that tells you that you can call it with `await`, you need to create the *path operation functions* that uses it with `async def`, like in:

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

More technical details

You might have noticed that `await` can only be used inside of functions defined with `async def`.

But at the same time, functions defined with `async def` have to be "awaited". So, functions with `async def` can only be called inside of functions defined with `async def` too.

So, about the egg and the chicken, how do you call the first `async` function?

If you are working with **FastAPI** you don't have to worry about that, because that "first" function will be your *path operation function*, and FastAPI will know how to do the right thing.

But if you want to use `async` / `await` without FastAPI, you can do it as well.

Write your own async code

Starlette (and **FastAPI**) are based on AnyIO [🔁], which makes it compatible with both Python's standard library asyncio [🔁] and Trio [🔁].

In particular, you can directly use AnyIO [🔁] for your advanced concurrency use cases that require more advanced patterns in your own code.

And even if you were not using FastAPI, you could also write your own async applications with AnyIO [🔁] to be highly compatible and get its benefits (e.g. *structured concurrency*).

I created another library on top of AnyIO, as a thin layer on top, to improve a bit the type annotations and get better **autocompletion**, **inline errors**, etc. It also has a friendly introduction and tutorial to help you **understand** and write **your own async code**: Asyncer [🔁]. It would be particularly useful if you need to **combine async code with regular** (blocking/synchronous) code.

Other forms of asynchronous code

This style of using `async` and `await` is relatively new in the language.

But it makes working with asynchronous code a lot easier.

This same syntax (or almost identical) was also included recently in modern versions of JavaScript (in Browser and NodeJS).

But before that, handling asynchronous code was quite more complex and difficult.

In previous versions of Python, you could have used threads or Gevent [🔁]. But the code is way more complex to understand, debug, and think about.

In previous versions of NodeJS / Browser JavaScript, you would have used "callbacks". Which leads to "callback hell".

Coroutines

Coroutine is just the very fancy term for the thing returned by an `async def` function. Python knows that it is something like a function, that it can start and that it will end at some point, but that it might be paused ⏸ internally too, whenever there is an `await` inside of it.

But all this functionality of using asynchronous code with `async` and `await` is many times summarized as using "coroutines". It is comparable to the main key feature of Go, the "Goroutines".

Conclusion

Let's see the same phrase from above:

> Modern versions of Python have support for **"asynchronous code"** using something called **"coroutines"**, with **`async` and `await`** syntax.

That should make more sense now. ✨

All that is what powers FastAPI (through Starlette) and what makes it have such an impressive performance.

Very Technical Details

Warning

You can probably skip this.

These are very technical details of how **FastAPI** works underneath.

If you have quite some technical knowledge (coroutines, threads, blocking, etc.) and are curious about how FastAPI handles `async def` vs normal `def`, go ahead.

Path operation functions

When you declare a *path operation function* with normal `def` instead of `async def`, it is run in an external threadpool that is then awaited, instead of being called directly (as it would block the server).

If you are coming from another async framework that does not work in the way described above and you are used to defining trivial compute-only *path operation functions* with plain `def` for a tiny performance gain (about 100 nanoseconds), please note that in **FastAPI** the effect would be quite opposite. In these cases, it's better to use `async def` unless your *path operation functions* use code that performs blocking I/O.

Still, in both situations, chances are that **FastAPI** will still be faster than (or at least comparable to) your previous framework.

Dependencies

The same applies for dependencies. If a dependency is a standard `def` function instead of `async def`, it is run in the external threadpool.

Sub-dependencies

You can have multiple dependencies and sub-dependencies requiring each other (as parameters of the function definitions), some of them might be created with `async def` and some with normal `def`. It would still work, and the ones created with normal `def` would be called on an external thread (from the threadpool) instead of being "awaited".

Other utility functions

Any other utility function that you call directly can be created with normal `def` or `async def` and FastAPI won't affect the way you call it.

This is in contrast to the functions that FastAPI calls for you: *path operation functions* and dependencies.

If your utility function is a normal function with `def`, it will be called directly (as you write it in your code), not in a threadpool, if the function is created with `async def` then you should `await` for that function when you call it in your code.

Again, these are very technical details that would probably be useful if you came searching for them.

Otherwise, you should be good with the guidelines from the section above: In a hurry?.