

Aug 11, 2019 | in Essays  python webdev asgi

 Edit

Introduction to ASGI: Emergence of an Async Python Web Ecosystem

If you were thinking Python had been getting locked into data science, think again! Python web development is back with an async spin, and it's exciting.



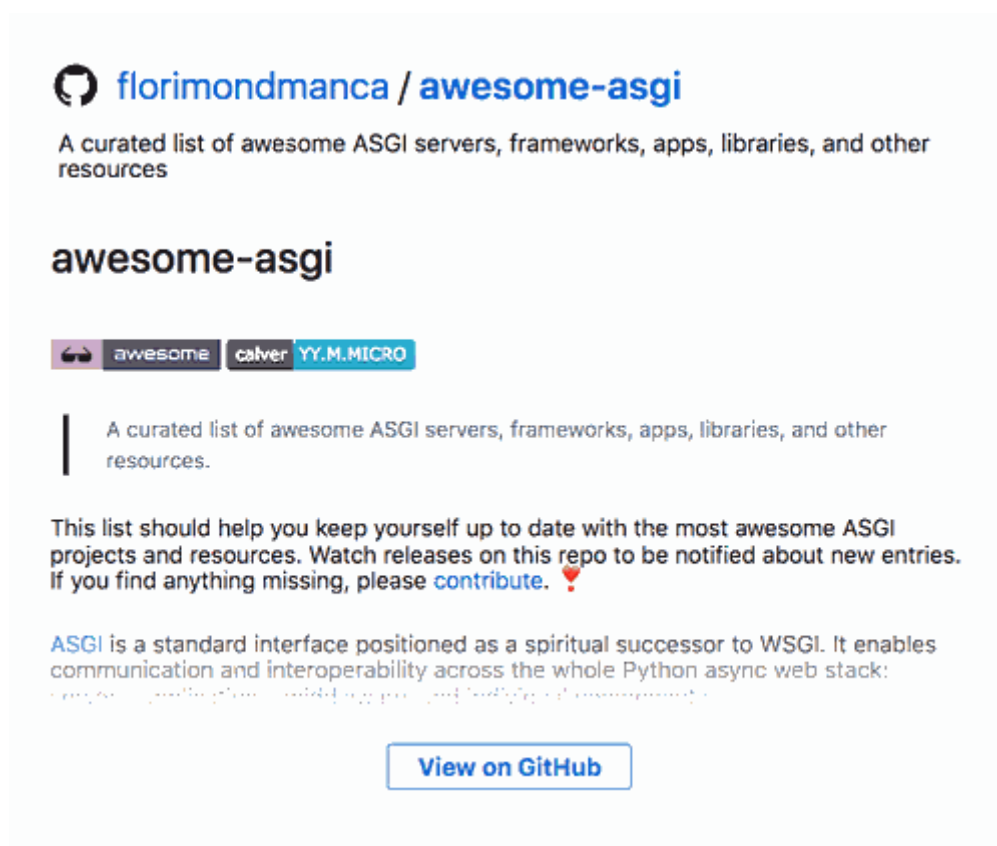
"Turtles on body of water", Ricard Baraham via unsplash.com

There's a lot of exciting stuff happening in the Python web development ecosystem right now — one of the main drivers of this endeavour is [ASGI](#), the Asynchronous Server Gateway Interface.

I already mentioned ASGI several times here, in particular when [announcing Bocadillo](#) and [tartiflette-starlette](#), but I never really took the time to write a thorough introduction about it. Well, here we are!

This post is targeted at people interested in recent trends of Python web development. I want to take you on a guided tour about **what ASGI is** and **what it means for modern Python web development**.

Before we begin, I'd like to announce that I recently created [awesome-asgi](#), an *awesome list* to help folks keep track of the ever-expanding ASGI ecosystem. 🙌



You can watch releases to be notified of new entries to the list. 👁

Alright, let's dive in!

It all started with `async/await`

Contrary to JavaScript or Go, Python is not a language that had asynchronous execution baked in from the start. For a long time, executing things concurrently in

Python could only be achieved using multithreading or multiprocessing, or resorting to specialized networking libraries such as eventlet, gevent, or Twisted. (Back in 2008, Twisted already had APIs for asynchronous coroutines, e.g. in the form of [inlineCallbacks](#) and [deferredGenerator](#).)

But this all changed in Python 3.4+. Python 3.4 [added `asyncio` to the standard library](#), adding support for cooperative multitasking on top of generators and the `yield from` syntax.

Later, the `async / await` syntax was [added in Python 3.5](#). Thanks to this, we now had **native coroutines** independent of the underlying implementation, which opened the gold rush towards Python concurrency.

And what a rush it was, indeed! Since 3.5 was released, the community has been literally **async-ifying all the things**. If you're curious, a lot of the resulting projects are now listed in [aio-libs](#) and [awesome-asyncio](#).

Well, you guessed it — this also means that **Python web servers and apps are moving towards async**. In fact, all the cool kids are doing it! ([Even Django](#).)

An overview of ASGI

Now, how does ASGI fit in all of this?

From a 1000-foot perspective, ASGI can be thought of as the glue that allows Python asynchronous servers and applications to communicate with each other. It shares a lot of design ideas with [WSGI](#), and is often presented as its spiritual successor with async built-in.

Here's what this mental model looks like in a diagram:



At a very high-level, ASGI is a communication interface between apps and servers.

But in reality, it's a bit more complex than that.

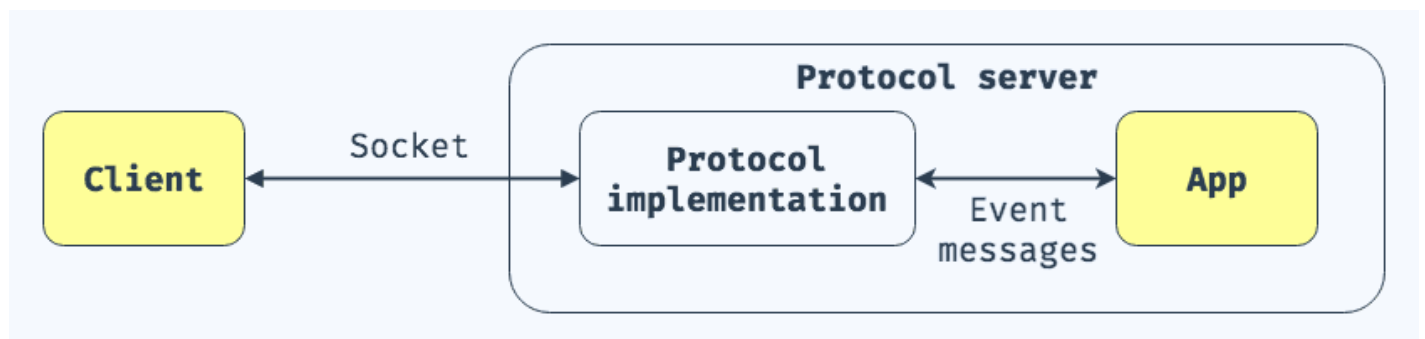
To find out how ASGI really works, let's take a look at the [ASGI specification](#):

ASGI consists of two different components:

- A *protocol server*, which terminates sockets and translates them into connections and per-connection event messages.
- An *application*, which lives inside a *protocol server*, is instantiated once per connection, and handles event messages as they happen.

So according to the spec, what ASGI really specifies is a [message format](#) and how those messages should be exchanged between the application and the protocol server that runs it.

We can now revise our diagram into a more detailed version:



*How ASGI **really** works.*

There are many more interesting details to look at, obviously. For example, you can take a look at the [HTTP and WebSocket specification](#).

Besides, although the spec focuses a lot on server-to-application communication, ASGI turns out to encompass *much more* than that.

We'll get to this in a minute, but first...

ASGI basics

Now that we've seen how ASGI fits in the Python web ecosystem, let's take a closer look at what it looks like in code.

ASGI relies on a simple mental model: when the client connects to the server, we instantiate an application. We then feed incoming bytes into the app and send back whatever bytes come out.

"Feed into the app" here really means *call the app* as if it were a function, i.e. something that takes some input, and returns an output.

And in fact, that's all an ASGI app is — a *callable*. The shape of this callable is, again, defined by the ASGI spec. Here's what it looks:

```
async def app(scope, receive, send):  
    ...
```

The signature of this function is what the "I" in ASGI stands for: an interface which the application must implement for the server to be able to call it.

Let's take a look at the 3 arguments:

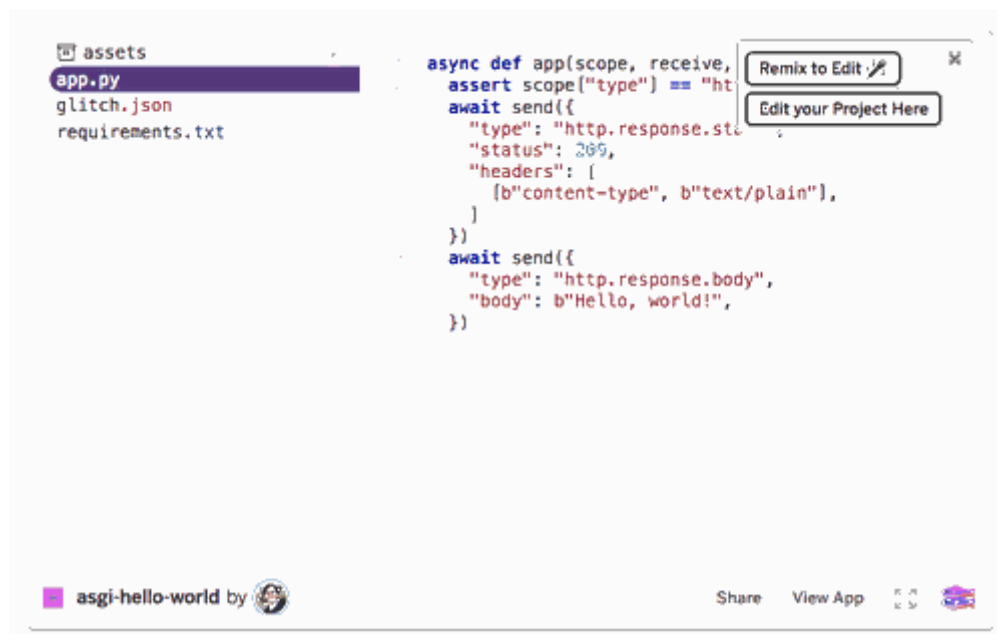
- `scope` is a dictionary that contains information about the incoming request. Its contents vary between HTTP and WebSocket connections.
- `receive` is an asynchronous function used to receive *ASGI event messages*.
- `send` is an asynchronous function used to send ASGI event messages.

In essence, these arguments allow you to `receive()` and `send()` data over a communication channel maintained by the protocol server, as well as know in what context (or `scope`) this channel was created.

I don't know about you, but the overall look and shape of this interface fits my brain really well. Anyway, time for code samples.

Show me the code!

To get a more practical feel of what ASGI looks like, I created a minimal project which showcases a raw ASGI HTTP app served by [uvicorn](#) (a popular ASGI server):



```
assets
app.py
glitch.json
requirements.txt

async def app(scope, receive,
assert scope["type"] == "ht
await send({
    "type": "http.response.sta
    "status": 200,
    "headers": [
        [b"content-type", b"text/plain"],
    ]
})
await send({
    "type": "http.response.body",
    "body": b"Hello, world!",
})
```

Here, we use `send()` to send an HTTP response to the client: we send headers first, and then the response body.

Now, with all these dictionaries and raw binary data, I'll admit that bare-metal ASGI isn't very convenient to work with.

Luckily, there are higher-level options — and that's when I get to talk about [Starlette](#).

Starlette is truly a fantastic project, and IMO a foundational piece of the ASGI ecosystem.

In a nutshell, it provides a toolbox of higher-level components such as requests and responses that you can use to abstract away some of the details of ASGI. Here, take a look at this Starlette hello world:

```
# app.py
from starlette.responses import PlainTextResponse
```

```
async def app(scope, receive, send):
    assert scope["type"] == "http"
    response = PlainTextResponse("Hello, world!")
    await response(scope, receive, send)
```

Starlette does have everything you'd expect from an actual web framework — routing, middleware, etc. But I decided to show this stripped-down version to hint you at the real power of ASGI, which is...

Turtles all the way down

The interesting and downright *game-changing* bit about ASGI is the concept of "[Turtles all the way down](#)", an expression originally coined (I think?) by Andrew Godwin, the person behind Django migrations and now the [Django async revamp](#).

But what does it mean, exactly?

Well, because ASGI is an abstraction which allows to tell in which context we are and to receive and send data *at any time*, there's this idea that ASGI can be used not only between servers and apps, but really *at any point in the stack*.

For example, the Starlette `Response` object is an ASGI application itself. In fact, we can strip down the Starlette example app from earlier to *just this*:

```
# app.py
app = PlainTextResponse("Hello, world!")
```

How *ridiculous* is that?! 🥰

But wait, there's more.

The deeper consequence of "Turtles all the way down" is that we can build all sorts of applications, middleware, libraries and other projects, and ensure that they will

be **interoperable** as long as they all implement the ASGI application interface.

(Besides, from my own experience building [Bocadillo](#), embracing the ASGI interface very often (if not *always*) results in much cleaner code.)

For example, we can build a ASGI middleware (i.e. an app that wraps another app) to display the time a request took to be served:

```
# app.py
import time

class TimingMiddleware:
    def __init__(self, app):
        self.app = app

    async def __call__(self, scope, receive, send):
        start_time = time.time()
        await self.app(scope, receive, send)
        end_time = time.time()
        print(f"Took {end_time - start_time:.2f} seconds")
```

To use it, we simply wrap it around an app...

```
# app.py
import asyncio
from starlette.responses import PlainTextResponse

async def app(scope, receive, send):
    await asyncio.sleep(1)
    response = PlainTextResponse("Hello, world!")
    await response(scope, receive, send)

app = TimingMiddleware(app)
```

...and it will magically *just work*.


```
$ uvicorn app:app
INFO: Started server process [59405]
INFO: Waiting for application startup.
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
...
INFO: ('127.0.0.1', 62718) - "GET / HTTP/1.1" 200
Took 1.00 seconds
```

The amazing bit about this is that `TimingMiddleware` can wrap *any* ASGI app. The inner app here is super simple, but it could be *a full-blown, real-life project* (think hundreds of API and WebSocket endpoints) — it doesn't matter, as long as it's ASGI-compatible.

(There's a more production-ready version of such a timing middleware: [timing-asgi](#).)

Why should I care?

While I think interoperability is a strong selling point, there are many more advantages to using ASGI-based components for building Python web apps.

- **Speed:** the async nature of ASGI apps and servers make them really fast (for Python, at least) — we're talking about 60k-70k req/s (consider that Flask and Django only achieve 10-20k in a similar situation).
- **Features:** ASGI servers and frameworks gives you access to inherently concurrent features (WebSocket, Server-Sent Events, HTTP/2) that are impossible to implement using sync/WSGI.
- **Stability:** ASGI as a spec has been around for about 3 years now, and version 3.0 is considered very stable. Foundational parts of the ecosystem are stabilizing as a result.

In terms of libraries and tooling, I don't think we can say we're there *just yet*. But thanks to a very active community, I have strong hopes that the ASGI ecosystem reaches feature parity with the traditional sync/WSGI ecosystem real soon.

Where can I find ASGI-compatible components?

In fact, more and more people are building and improving projects built around ASGI. This includes servers and web frameworks obviously, but also middleware and product-oriented applications such as [Datasette](#).

Some of the non-web framework components that I got the most excited about are:

- [Mangum](#): ASGI support for AWS Lambda
- [datasette-auth-github](#): GitHub authentication for ASGI apps
- [tartiflette-starlette](#) (I wrote this one!): ASGI support for Tartiflette, an async GraphQL engine.

While seeing the ecosystem flourish is great, I've personally been having a hard time keeping up with everything.

That's why, as announced at the beginning of this article I created [awesome-asgi](#). My hope is that it helps everyone keep up with all the awesome things that are happening in the ASGI space. (And seeing that it almost reached 100 stars in a few days, I have a feeling there was indeed a need to colocalize ASGI resources.)

Wrapping up

While it might look like an implementation detail, I truly think that ASGI has laid down the foundations for a new era in Python web development.

If you want to learn more about ASGI, take a look at the various [publications](#) (articles and talks) listed in [awesome-asgi](#). To get your hands dirty, try out any of the following projects:

- [uvicorn](#): ASGI server.
- [Starlette](#): ASGI framework.

- [TypeSystem](#): data validation and form rendering
- [Databases](#): async database library.
- [orm](#): asynchronous ORM.
- [HTTPX](#): async HTTP client w/ support for calling ASGI apps (useful as a test client).

These projects were built and are maintained by Encode, which mostly means Tom Christie. There are open discussions on setting up an [Encode maintenance team](#), so if you were looking for an opportunity to help advance an open source niche, there you go!

Enjoy your ASGI journey. ❤️

1 Comment - powered by [utteranc.es](#)

Kludex commented 2 months ago

Thank you! :)

Write

Preview

Sign in to comment

 Styling with Markdown is supported

Sign in to comment

Links

[GitHub](#) [Twitter](#) [LinkedIn](#) [DEV](#)

© 2021 Florimond Manca · [Source](#)