A Framework of One's Own

June 2026

7 min read • 1,625 words

In October 2018 I released a web framework called Responder, and to understand why it exists you have to remember what the Python web felt like that year, because the web it was born into no longer exists.

It was the end of the synchronous era and nobody had told the frameworks yet. Flask and Django ruled, both built on WSGI, both fundamentally blocking, both excellent and both showing their age. `async` / `await` had landed in the language a couple of years earlier, but the async web ecosystem was a frontier town: ASGI was a young spec, Starlette and uvicorn were fresh out of Tom Christie's workshop, and writing an async service meant choosing between power tools with no handles. Meanwhile the industry was deep in its microservices-and-GraphQL period, everything was becoming an API, and HTTP/2 server push was going to change everything.

> It did not change everything. Browsers eventually deprecated server push entirely, and a few months ago I deleted it from Responder's backlog, which is its own little memento mori: frameworks age not only by their own decay but by the web rotting out from under their assumptions.

I had spent most of a decade on the other side of the protocol, asking one question about HTTP clients: why is this harder than the way people think? Requests was the answer, and by 2018 it was everywhere. But serving HTTP still felt like a different religion from consuming it, with different gods and a different liturgy, and that asymmetry offended me. The same protocol. The same developer. Why two mental models?

So Responder flipped Requests inside out:

```
import responder

api = responder.API()

@api.route("/")
def home(req, resp):
    resp.html = "<h1>Hello, world.</h1>"

@api.route("/hello/{name}")
async def hello(req, resp, *, name):
    resp.media = {"greeting": f"Hello, {name}!"}

api.run()
```

`resp.text` sends text. `resp.html` sends HTML. `resp.media` sends JSON. Case-insensitive headers, like Requests. The `async` keyword is optional per-handler, sync and async living in the same app, because the right time to pay the async tax is when you need it and not one minute before. Background tasks, WebSockets, OpenAPI docs, and templates in the box. The niceties of Flask, the performance philosophy of Falcon, and the ergonomics of Requests, in one place.

Two months later FastAPI arrived, built on the same Starlette foundation, took the typed-routes-and-OpenAPI thread and ran it to the moon. It deserved the win; if you're building a production API with a team, it's what I'd hand you. Several ideas that Responder carried early, automatic async handling, type-aware serialization, docs generated from the code, became table stakes across the ecosystem. Being ahead of your time and being beaten to the market are, it turns out, frequently the same event viewed from different chairs.

And here I have to be honest about my own role: I never really marketed Responder. Not then, not since. I'd shout about it for a week, then go quiet for a year. The README never had a growth strategy. Some of that was what the era was doing to me personally, and some of it was a lesson Requests had already taught me at great cost: a tool with millions of users stops being a tool and

becomes an institution you owe rent to. I don't think I ever wanted Responder to be big. I think I wanted it to be right, which is a different ambition with a different finish line.

The Freeze

Here's what the release history actually looks like, and I'm sharing it because solo open source is full of timelines like this that nobody publishes.

Version 2.0.7 shipped in January 2021. The next release, 3.0.0, shipped in March 2026.

Five years of silence. Not abandonment, exactly; the repository sat there, issues accumulated politely, the code still worked. But every path forward ran through the same wall: modernization. New Python versions, Starlette drift, packaging upheaval, typing expectations, CI rot, documentation debt. None of it is hard, exactly. All of it together is a mountain of exactly the kind of work that has no dopamine in it anywhere, and a solo maintainer pays for that mountain out of the same budget he uses to stay alive. Mine was overdrawn for most of that period. This is the development hump that kills more small projects than any technical failure: not a bug, not a design flaw, just five years of deferred chores standing between the maintainer and anything fun.

The Hump Breaks

Then, on March 22, 2026, Responder shipped seven releases in one day.

Version 3.0.0 in the morning. Versions 3.1 through 3.4 across the afternoon and evening, a hundred-some commits clearing the whole mountain: modern packaging, modern Starlette, the test suite rebuilt, the docs overhauled. Two days later, 3.5 and 3.6 brought something genuinely new, structured logging with per-request context, request IDs, access timing, the kind of feature you add when a framework is alive. The releases since then read like a healthy project's changelog: a race condition fixed in the rate limiter, a memory leak closed, blocking file I/O made properly async, an XSS hole patched, the boring diligent engineering that quietly never happened for half a decade. And on the day this essay went out, versions 3.7 through 3.12 shipped within hours of each other:

dependency injection for route handlers and WebSockets, per-route rate limits, handlers that can simply return a value, OpenAPI 3.1, content-negotiated errors, ETags, request streaming, range requests and resumable downloads, request timeouts, server-side sessions, Prometheus metrics, a stack of routing bugs nobody had touched in years. The feature that replaced server push in the backlog is real now. Seven releases and the essay about the freeze, all on the same day. I keep having to revise this paragraph upward, which is the most cheerful editing problem I have ever had.

The same day the hump broke, this site started running on Responder, ported from Flask in a single afternoon. That was not a coincidence. Both were the same event: AI collaboration arrived at the level where the unfunded chores became conversations.

I've written about the workflow that makes this work: write the foundation by hand, then let AI extend it faithfully. Responder is the purest case I own. The 2018 codebase is hand-written taste all the way down, every API decision deliberate, the whole "for humans" philosophy compressed into working software. When Claude reads it, it reads intent, and it extends that intent, in my voice, at a pace no exhausted solo maintainer could match. The mountain of chores that cost five years of avoidance cost a few long sessions of directed collaboration. I supplied the taste and the judgment and the no-that's-not-how-we-do-it. The AI supplied the part I could never reliably supply: the stamina.

There's a sentence I keep arriving at from different directions this year, and here it is again: the bottleneck was never the thinking. For writing, the bridge between knowing and saying. For this framework, the bridge between caring and maintaining. AI didn't give Responder new ideas. It gave the old ideas a way out of the freezer.

A Userbase of Approximately One

So what is Responder now? Here's the honest accounting, and it's the part I'm fondest of.

It's a framework with one heavy user, and you're reading his website through it right now. Every page on kennethreitz.org is served by a single `engine.py`, thirty-some routes, markdown rendering, image galleries, PDF export, OG

images, RSS, search, all of it running on the framework I built in 2018, on a server named Mercury that lives in a rack. When the framework needs a feature, its user requests it directly, in the sense that I talk to myself. When a release ships, its user upgrades the same hour. The feedback loop isn't tight; it's closed.

The industry has a derisive shrug for this: no adoption, no community, hobby project. I spent years inside that value system and I'd like to offer the correction from the far side of it. I've already written about what chasing the big userbase cost me. Requests serves thirty-some million installs a day and I carried it like a piano on my back. Responder serves, as far as I can tell, mostly me, and it is the most uncomplicated joy in my software life. It's infrastructure for one, the same economics that AI quietly flipped for everything else: a tool no longer needs a market to justify its maintenance, because maintenance no longer costs what it used to. It needs a user it fits. I am extraordinarily well fit.

And because it's mine and small, it gets to stay opinionated in ways a big framework never could. No roadmap committee. No deprecation-policy negotiations. No marketing cadence, which is good, because I've demonstrated for eight years that I don't have one. Just a tool, a craftsman, a maintainer who is the entire interface, and a website that exercises every feature daily. If someone else finds it and loves it, the door's open and the docs are genuinely good now. But it doesn't need you, which I've come to believe is the most relaxing property software can have, for everyone involved.

Virginia Woolf said the writer needs a room of her own. I'd extend it: the builder needs a framework of his own. Something nobody else's deadlines live in. Something that fits your hand because it is your hand, where the whole stack from the route decorator to the rack in the datacenter answers to one taste. Requests was for everyone, and everyone got it, and that was the right call and I paid for it.

Responder was for me. It took me eight years to realize that was the feature.

The best products ship to a userbase of one. The trick is making sure the one is happy. He is.