This Site Now Runs on Responder

March 2026

6 min read • 1,432 words

As of today, kennethreitz.org runs on Responder, my own web framework. Not Flask. Not FastAPI. The framework I built in 2018 as an experiment in making the server side feel like the client side.

The port took a single session. One afternoon. Me and Claude Opus 4.6, reading every route, rewriting every handler, testing every endpoint, deploying to production. The whole thing.

I want to talk about why, and what it felt like, and what broke along the way.

What Responder Is

For those who haven't encountered it: Responder is a web framework built on Starlette that flips Requests inside out. If Requests is how you consume HTTP, Responder is how you serve it, using the same mental model.

```
import responder

api = responder.API()

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

@api.route("/api/data")
def data(req, resp):
    resp.media = {"message": "Hello from Responder"}

if __name__ == "__main__":
    api.run()
```

`resp.text` sends text. `resp.html` sends HTML. `resp.media` sends JSON. The async keyword is optional. Case-insensitive headers, just like Requests. It was ahead of its time when I built it. Some of the ideas, automatic async handling and type-aware serialization, showed up later in FastAPI. Responder was always more about API design philosophy than market share. Now it's on version 3.2, and it's serving the page you're reading.

Why Port from Flask

The site was running on Flask, wrapped in ASGI via asgiref. Sync code pretending to be async. It worked fine. Flask is excellent software and I have nothing but respect for it.

But there's a principle in software development called dogfooding: use your own tools. If you build a web framework and don't run your own site on it, what are you really saying about your confidence in it?

> The term "dogfooding" comes from the idea that a dog food company should eat its own product. It's crude but accurate. If the builder won't use it, why should anyone else?

This site is not a toy project. It serves 300+ markdown files, image galleries, PDF export, OG image generation, RSS feeds, full-text search, and 25+ routes. If Responder can handle all of that in production, it's ready for real work. If it can't, I need to know that too.

The honest answer is both practical and personal. Practical: native ASGI is cleaner than sync-wrapped-in-async. Personal: there's something right about the loop closing. I built Responder during what I've called the Lego Bricks era of open source. Years later, it's serving my own site. The framework matured. The site matured. The builder matured.

How AI Made a Same-Day Port Possible

Let me be specific about what the port involved, because "I ported my site" can sound trivial.

The Flask version had multiple blueprint files, each containing a subset of routes. Template rendering relied on Flask automatically injecting `request` and `config` into every template context. Routes used Flask's decorator syntax, response patterns, redirect semantics. The whole thing was wired together in a way that assumed Flask would be there forever.

Porting meant: reading every blueprint file, understanding every route handler, translating Flask idioms to Responder patterns, rewriting template rendering, handling all the places where Flask's implicit behavior needed to become explicit, testing every endpoint, and fixing the things that broke.

A solo developer would budget days for this. Maybe a week if they were being careful. Claude read the entire codebase, understood the architecture, and wrote the port. I directed. Claude executed. Both of us caught problems. It was genuine pair programming, the kind I wrote about when discussing how AI collaboration works on the KJV Study project.

The result is a single `engine.py` file. All routes in one place. No blueprints, no scattered files. Simpler.

```
"""Responder-based engine for kennethreitz.org."""

import responder

api = responder.API(
    templates_dir="tuftecms/templates",
    static_dir="tuftecms/static",
    static_route="/static",
)

@api.route("/")
async def homepage(req, resp):
    blog_data = get_blog_cache()
    recent_posts = blog_data.get("posts", [])[:6]
    resp.html = render("homepage.html", req, "/",
        title="Home",
        recent_posts=recent_posts,
    )
```

Clean. Familiar. The response object works the way your brain expects it to.

What Broke

Three things broke, and all three were instructive.

Template context injection. Flask automatically injects `request` and `config` into every Jinja2 template. Responder doesn't. Every template on the site referenced `request.path` for navigation highlighting and `config` for environment variables. The fix was a `render()` helper function that wraps Responder's template engine and injects Flask-compatible shims:

```
class RequestWrapper:
    """Wraps Responder request to provide Flask-like interface."""
    def __init__(self, req, path):
        self._req = req
        self.path = path
        self.environ = {}

class FakeConfig(dict):
    """Minimal config stand-in for Flask template compatibility."""
    def get(self, key, default=None):
        return os.environ.get(key, default)

def render(template, req, path="/", **kwargs):
    """Render a template with common context."""
    kwargs["request"] = RequestWrapper(req, path)
    kwargs["config"] = _config
    return api.templates.render(template, **kwargs)
```

Not elegant. Effective. The templates didn't need to change at all.

Route priority. The site has a catch-all route that serves markdown files from any path. In Flask, static file serving happened through a separate mechanism that took priority. In Responder, the catch-all route was intercepting requests for static files, CSS, JavaScript, images, everything. The fix was simple but non-obvious: define all specific routes before the catch-all, and add an explicit static file pass-through. The comment in `engine.py` now reads:

```
# IMPORTANT: All specific routes must be defined BEFORE the catch-all route.
```

Every production framework teaches you something that toy examples never will. This is one of those lessons.

Response patterns. Flask returns strings or `Response` objects from route handlers. Responder mutates the `resp` object in place. Every `return render_template(...)` became `resp.html = render(...)`. Every `return jsonify(...)` became `resp.media = {...}`. Every `redirect(url)` became `api.redirect(resp, url)`. Mechanical but pervasive. Exactly the kind of tedious, high-accuracy transformation that AI handles better than humans.

What Got Better

Not everything was a compatibility shim. Some things genuinely improved.

The site is now async-native. Not sync code wrapped in ASGI adapters. Actual async handlers running on Starlette's event loop. GZip middleware, session middleware, trusted host middleware all come free from Starlette's middleware stack. The dependency on asgiref is gone.

The architecture is simpler. Four blueprint files collapsed into one engine file. The mental overhead of "which blueprint handles this route?" is gone. You open `engine.py`, you see everything. For a personal site, this is the right level of complexity.

And Responder itself got tested against a real workload. I've already found two things I want to improve in the framework based on this port. That's the whole point of dogfooding. You don't find the rough edges in your workshop. You find them in the field.

The Loop Closes

In The Lego Bricks Era, I wrote about the golden age of open source, when building beautiful tools and sharing them was the point. I built Responder during that era. It was an experiment in API design, an attempt to answer: what if the server side felt as natural as the client side?

Years passed. The framework sat. I maintained it, updated it, but I didn't use it for anything that mattered to me personally. Meanwhile, my site ran on Flask, which is fine, but which was always someone else's tool.

There's something satisfying about the loop closing. The same person who built the framework now relies on it daily. The same philosophy that created Requests, that HTTP should feel natural and human, now serves every page on this site through Responder. The tool serves the builder. The builder improves the tool. The improvements serve everyone who uses it.

This connects to what I keep coming back to: the recursive loop between code and consciousness. The tools we build shape how we think. Using my own tool shapes how I think about what the tool should be. Which shapes the tool. Which shapes everyone who uses it. The loop doesn't end. It just keeps turning.

I wrote in that Lego Bricks essay that tech became craft instead of lifestyle for me. This is what craft looks like in practice. Not building something new for the sake of building. Using what you've already built, maintaining it, improving it based on real use, and letting it serve the life you're actually living.

The site you're reading was built by the same person who built the framework it runs on, ported in an afternoon with the help of an AI that understood both codebases, and deployed to production the same day. No drama. No weeks of migration planning. Just a builder using his own tools.

That's how it should work.

This site runs on Responder, served from a single `engine.py`. For the framework's philosophy, see From HTTP to Consciousness. For the open source era that produced it, see The Lego Bricks Era. For site architecture details, see the Colophon.