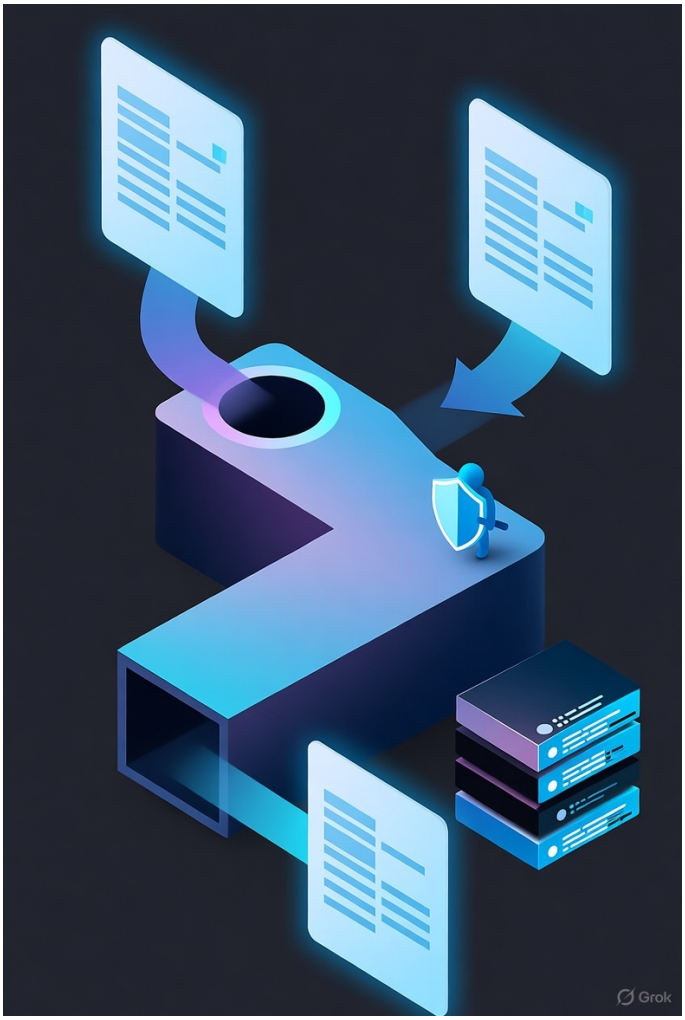
Introducing OpenAPILoggingMiddleware



Introduction

In the world of **OpenAPI-driven Swift servers**, visibility is everything. You can't fix what you can't see — and you can't optimize what you don't understand.

When I started building modular, OpenAPI-first architectures in Swift, one thing stood out: **logging was either overwhelming or absent**. Either you got a raw dump of everything (headers, binary data, and stack traces), or you got nothing at all. Neither is particularly helpful when debugging or tracing an issue across microservices.

Frameworks like **OpenAPIRuntime** and **OpenAPIVapor** provide a strong foundation, but they deliberately avoid opinions about logging. They leave that up to you. And that's where most developers end up adding scattered `print()` statements or ad-hoc `logger.info()` calls inside handlers just to see what's going on.

That's not clean architecture. It's noise.

So I built **OpenAPILoggingMiddleware**, a small, composable middleware that gives you *just enough visibility* — not too much, not too little — when working with OpenAPI-based servers.

The Problem: Logging in a Generated World

OpenAPI code generation has revolutionized how backend systems are written. You define your contracts, generate both server and client, and everything should “just work.” But once requests start flowing, visibility often becomes a blind spot.

If you’ve ever tried debugging an OpenAPI-generated Swift server, you might recognize these problems:

> Middleware logs every byte of the body, even binary data.
> Multi-part and SSE requests produce unreadable noise.
> Response logs are buried under system output.
> There’s no consistent way to see request duration or endpoint performance.
> You can’t easily toggle verbosity levels or redact sensitive data.

You end up either drowning in log output or adding `print` calls just to survive the debugging process.

This middleware solves that.

The Solution: A Lightweight Logging Layer

`OpenAPILoggingMiddleware` sits quietly in the pipeline — between your generated routes and the transport layer. It listens to every incoming request and outgoing response, measures duration, and prints a **structured summary** of what happened.

At its core, the idea is simple:

> Log exactly what you’d want to see if you were reading someone else’s code.

Here’s how it looks in your Vapor or OpenAPI setup:

```
import OpenAPILoggingMiddleware

app.middleware.use(OpenAPILoggingMiddleware(level: .info))
```

That one line transforms your debugging experience.

Under the Hood

The middleware implements the `OpenAPIMiddleware` protocol, which lets it intercept the lifecycle of each OpenAPI operation — before the request is handled and after the response is sent.

Here’s a conceptual overview:

```
public struct OpenAPILoggingMiddleware: OpenAPIMiddleware {
    public func intercept(
        _ request: Request,
        operationID: String,
        next: (Request) async throws -> Response
    ) async throws -> Response {
```

```
        let start = Date()
        let response = try await next(request)
        let duration = Date().timeIntervalSince(start) * 1000

        print("[\(request.method)] \(request.url.path) ->
\(response.status.code) [\(Int(duration)) ms]")
        return response
    }
}
```

Of course, the real implementation handles errors, structured formatting, and configurable verbosity, but the essence is the same: it's a *transparent layer of introspection* that doesn't alter your data flow.

Logging Philosophy

There's a fine balance between too much and too little information.

Traditional logging tools either **dump everything** (which nobody reads) or **show too little** (which leaves you guessing). `OpenAPILoggingMiddleware` follows a *minimalist* principle inspired by Apple's own frameworks — log what matters, hide what doesn't.

It's designed around three key ideas:

1. **Contextual logging** — every entry includes method, path, status, and duration.
2. **Progressive verbosity** — higher log levels show headers and bodies.
3. **Human readability first** — logs are meant to be scanned, not parsed by machines.

Here's an example of what a single request looks like in `.info` mode:

```
[POST] /mocks/messages/agent-responses (200 OK)
Duration: 123 ms
Body: {"message":"You shall not pass!"}
```

And at `.debug` level, you get more detail:

```
[POST] /mocks/messages/agent-responses
Headers:
  Content-Type: application/json
  Authorization: Bearer Gandalf-Was-Here-1
Duration: 123 ms
Body: {"message":"You shall not pass!"}
```

Readable. Structured. Useful.

Why Not Just Use Vapor's Logger?

That's a valid question.

Vapor already includes a powerful logging system based on SwiftLog, so why add another layer? Because Vapor's logger is **request-agnostic** — it's not aware of OpenAPI operations, generated endpoints, or typed models.

`OpenAPILoggingMiddleware` is **contract-aware**. It sits in the OpenAPI pipeline, so it can access the operation ID, typed body, and structured response — all without touching your route handlers. That distinction is crucial for OpenAPI-based architectures, where most of your server logic is generated.

Integration Examples

Adding it to a Vapor-based OpenAPI server is trivial:

```
import Vapor
import OpenAPIRuntime
import OpenAPIVapor
import OpenAPILoggingMiddleware

let app = try Application(.detect())

let server = try! MyOpenAPIServer(app: app)

app.middleware.use(OpenAPILoggingMiddleware(level: .info))
try app.run()
```

If you're using `OpenAPIRuntime` directly (without Vapor), it's equally simple:

```
var server = try OpenAPIServer()
server.middleware.append(OpenAPILoggingMiddleware(level: .debug))
try server.start()
```

No configuration files. No dependencies. It just works.

Extensibility and Customization

Logging needs vary between environments. In development, you might want to see every detail. In production, you want concise summaries. The middleware supports multiple log levels, allowing you to tailor verbosity to your needs:

- `.error` — log only failed requests.
- `.info` — log all requests with method, path, duration, and status.
- `.debug` — include headers and bodies.

In future releases, I plan to add **filters** and **formatters** — so you could, for instance, redact specific headers (`Authorization`, `Cookie`) or export logs in JSON format for ingestion into a centralized system.

Example:

```
let middleware = OpenAPILoggingMiddleware(
    level: .debug,
    redactHeaders: ["Authorization", "Cookie"]
)
```

Design Details: Why Simplicity Wins

Many logging libraries start small and end up as frameworks. They introduce dependency graphs, adapters, output sinks, configuration DSLs — and complexity sneaks in through the back door.

I deliberately avoided that path.

`OpenAPILoggingMiddleware` does one thing: **it shows what your OpenAPI server is doing**. Nothing else.

No JSON serialization frameworks. No dependency injection. No external configuration. Just clean Swift and a single dependency on `OpenAPIRuntime`.

The codebase is small enough to read in one sitting — and that's intentional. In my opinion, **you should be able to understand every line of code that runs in your server's core path**.

Performance Considerations

Logging always comes at a cost, but the impact here is minimal. The middleware measures request duration using `Date().timeIntervalSince(start)`, which is negligible compared to network latency or I/O.

You can safely keep it enabled even in staging or pre-production environments. In production, switching to `.error` mode will keep your logs clean while still providing visibility into failures.

Testing and Debugging

The middleware is fully testable using Vapor's `Application` test harness or plain `XCTest` with mock requests. Here's a simple test that validates duration and structure:

```
func testMiddlewareLogsRequestAndResponse() async throws {
    let app = Application(.testing)
    app.middleware.use(OpenAPILoggingMiddleware(level: .info))
    try await app.test(.POST, "/ping", afterResponse: { res in
        XCTAssertEqual(res.status, .ok)
    })
}
```

You can also inject your own logger conforming to `LogHandler` if you prefer structured output rather than printing to stdout.

Future Work

The roadmap includes:

1. **JSON Formatter** — for structured logs in production environments.
2. **Redaction Rules** — for headers and sensitive body fields.
3. **Metrics Hooks** — integration with Swift Metrics or Prometheus.
4. **Pluggable Output Destinations** — allowing streaming to files or external monitoring systems.
5. **Async Logging** — offloading logging I/O to background tasks for ultra-high performance scenarios.

Each of these will follow the same guiding principles: clarity, composability, and minimalism.

Philosophy: Clean Visibility

Logging is not just a developer tool — it's part of the interface between humans and systems. A well-designed logging layer doesn't scream for attention; it quietly reveals how the system behaves.

When I was designing this package, I thought about the elegance of Apple's own frameworks — the way their APIs feel inevitable, obvious in hindsight. That's what I aim for here: **a logging middleware that feels invisible until you need it, and indispensable once you use it.**

Example Output in Context

Here's a real-world example of multiple concurrent requests hitting the same endpoint:

```
[GET] /user/profile (200 OK)
Duration: 87 ms

[POST] /mocks/messages/agent-responses (200 OK)
Duration: 121 ms
Body: {"message":"You shall not pass!"}

[PATCH] /user/preferences (204 No Content)
Duration: 96 ms
```

Notice how easy it is to see patterns — which endpoints are slow, which ones are frequent, which ones failed. That's the essence of **observability** — not more data, but *useful* data.

Real-World Use

I use `OpenAPILoggingMiddleware` in all my OpenAPI projects — from small prototypes to complex SSE (Server-Sent Events) systems. It's particularly valuable when debugging **streamed responses** or **multipart form uploads**, where conventional logs become unreadable.

Because it's a simple `OpenAPIMiddleware`, it works with any generated server conforming to the OpenAPI ecosystem — including custom transports and pure SwiftNIO backends.

Closing Thoughts

This middleware is a reminder that sometimes the simplest tools make the biggest difference. It's easy to underestimate the power of **well-designed visibility** — until you remove it and start guessing again.

Whether you're debugging mock routes, profiling API latency, or simply curious about what your server is doing, `OpenAPILoggingMiddleware` gives you that quiet transparency every developer deserves.

? **Codeberg:** Mihaela/OpenAPILoggingMiddleware

"Clean code should be composable, testable, and visible when it runs."