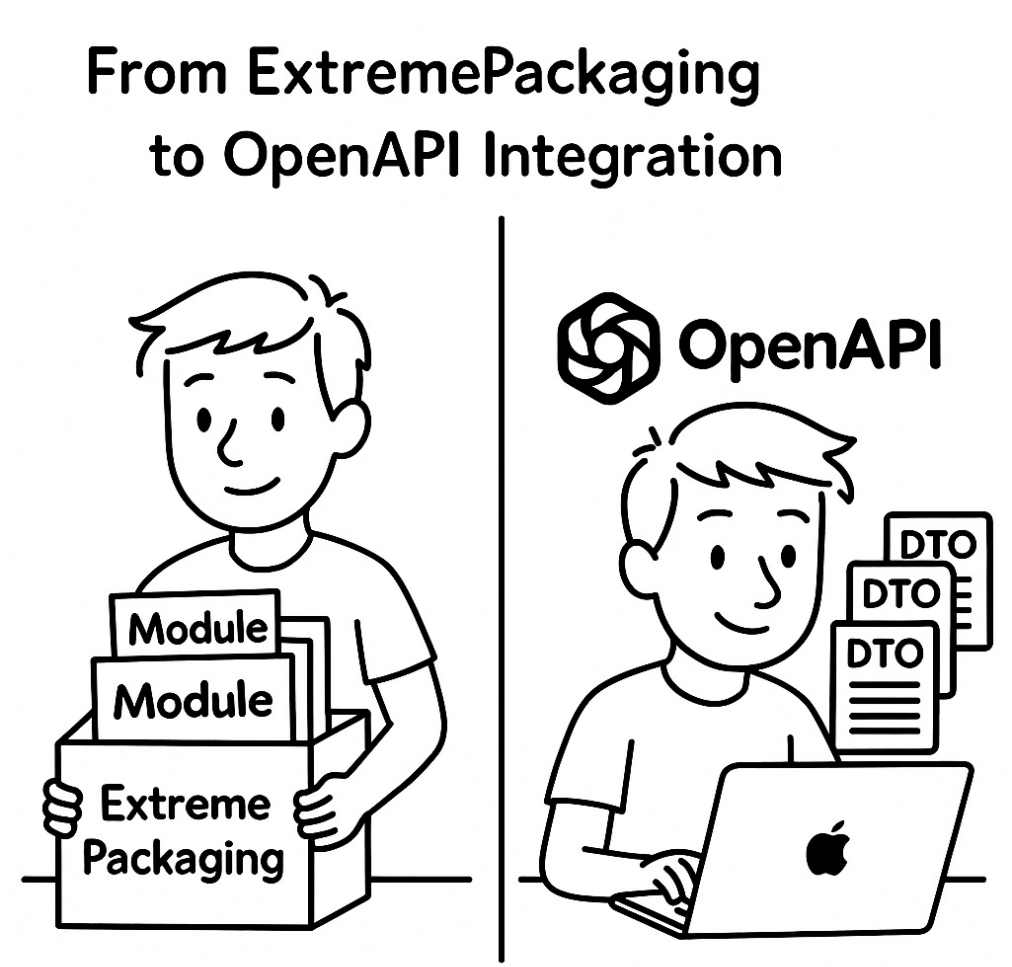
From ExtremePackaging to OpenAPI Integration



When I first published ExtremePackaging, the goal was simple — to prove that **highly modular Swift architectures** can scale across platforms while keeping build times fast, dependencies isolated, and the mental model crystal clear.

But then came the next step: *Could this same architecture host a complete OpenAPI workflow — with generated clients, servers, and full middleware integration — without losing its elegance?*

That's how the new reference implementation, **swift-openapi-extremepackaging-example**, was born.

? The Foundation — ExtremePackaging

The original ExtremePackaging repository introduced a clean, layered structure:

Shared packages for models and protocols
Independent feature modules for UI, data, and networking
No cross-package leaks
Unified Xcode workspace that felt like a monolith but built like microservices

The guiding philosophy: *Each feature should be an island, communicating only through well-defined contracts.*

That foundation made it ideal for integrating OpenAPI-generated code — which naturally fits into modular boundaries like `SharedApiModels`, `ApiClient`, and `ApiServer`.

? Evolving Toward OpenAPI

The OpenAPI version added an entire new layer of automation and functionality — transforming a static architecture into a **living, self-describing API ecosystem**.

1. OpenAPI Schema & Code Generation

At the heart of the project is the **OpenAPI specification (**`openapi.yaml`**)** — defining endpoints, models, and responses for a DummyJSON-compatible API.

Newly introduced elements:

? Full schema definitions for `Users`, `Posts`, `Products`, `Todos`, and `Carts`
? Error schemas (404, validation, authentication)
?? Integration with **Swift OpenAPI Generator**
? Automatic generation of clients, models, and server stubs
? Separation of generated code into `SharedApiModels`

This turned the architecture into a self-contained API ecosystem — one that can **generate, serve, and consume** its own endpoints.

---

2. API Server Implementation

A complete **Vapor-based local server** (`ApiServer`) was added to simulate real-world backend behavior:

17 endpoints fully implemented from the OpenAPI spec
Realistic mock data mirroring DummyJSON
Pagination and validation logic
Centralized error responses
?? Prefixed logging for easy tracing in the console

The server runs locally at `http://localhost:8080`, serving as both a mock backend and a test harness for the generated client.

---

3. Enhanced API Client Architecture

The client evolved from a simple abstraction into a **fully concurrent, actor-based networking layer**.

Highlights

- `ApiClient` actor manages shared state safely across async contexts
- Middleware chain introduced: **Logging ? Authentication ? Correction**
- Runtime environment switching between `.production`, `.local`, and `.mock`
- Shared singleton `ApiClientState` stores token, settings, and preferences

Example flow:

```
try await ApiClient.initializeShared(environment: .production)
let client = ApiClient.shared!
let auth = try await client.login(username: "emilys", password: "emilyspass")
await ApiClient.setToken(auth.accessToken)
let users = try await client.getUsers(limit: 10)
```

A clear separation between environment configuration and runtime state ensures deterministic, thread-safe behavior.

---

4. Middleware Integration

The client leverages two reusable middlewares from sibling packages:

OpenAPILoggingMiddleware

Provides structured, console + JSON logging with full request/response capture.

BearerTokenAuthMiddleware

Manages JWT token injection with a concurrency-safe actor and public operation rules.

Together, they demonstrate the power of **middleware chaining** in OpenAPI Runtime — clean, modular extensions without inheritance or global state.

---

5. YAMLMerger — The Key to Structured API Specs

The project uses **YamlMerger** — a Swift package that merges multiple YAML files into a single combined OpenAPI specification. If your project doesn't already include an `openapi.yaml`, YamlMerger ensures you have one — and helps you maintain a **structured, predictable folder layout** under `Tests/` or `Sources/SharedApiModels/schemas/`.

? Why It Must Be Copied into the Project

YamlMerger cannot simply be added as a SwiftPM dependency for build-time merging because of **SPM's read-only resolution model**:

1. Swift Package Manager stores dependencies in a **cached, read-only** location (`.build/checkouts/`).
2. The OpenAPI generator, however, needs **write access** to output the merged `openapi.yaml` file directly into your source tree.
3. SPM build scripts are not allowed to write to source folders outside their sandboxed build

directory.

? **Solution:** Copy the YamlMerger executable directly into your project (e.g. `Tools/YamlMerger/`) and call it from a pre-build script or CI pipeline. This guarantees write permissions and makes the tool available to everyone checking out the repo.

? What It Does

YamlMerger scans subdirectories (01 ? 08) and merges YAML fragments in deterministic order:

1. Folders are processed numerically.
2. `__*.yaml` files merge first within each folder.
3. Remaining files merge alphabetically.
4. The final output is a complete OpenAPI spec, suitable for Swift OpenAPI Generator.

? Example Schema Layout

```
Schema/
??? 01_Info/
??? 02_Servers/
??? 03_Tags/
??? 04_Paths/
??? 05_Webhooks/
??? 06_Components/
??? 07_Security/
??? 08_ExternalDocs/
```

Each folder corresponds to a section of the OpenAPI spec, allowing multiple developers to work on different endpoints, schemas, or components without conflicts.

?? Typical Workflow

```
# Merge schemas before build
./Tools/YamlMerger merge   --input Sources/SharedApiModels/schemas/   --output
Sources/SharedApiModels/openapi.yaml
```

You can run this manually, in a pre-build phase, or as part of CI/CD automation.

? Pro Tip

If your project starts **without** an `openapi.yaml`, placing schema fragments in structured folders under `Tests/` ensures your API structure remains organized — even before full code generation. YamlMerger gives your tests (and your teammates) a **shared, visual map** of your API's evolving shape.

---

6. Test Coverage Expansion

Two new test suites validate both local and production APIs:

? `ApiClientLocalTests.swift` — 25 tests targeting the local Vapor server
? `ApiClientProductionTests.swift` — 29 integration tests against DummyJSON API

Tests cover:

- Authentication and token persistence
- Pagination behavior
- Error responses and invalid IDs
- Concurrent request handling

Together they form a **54-test safety net** proving both architecture and OpenAPI compliance.

? Architecture Snapshot

```
Packages/
??? Sources/
?   ??? ApiClient/
?   ??? ApiServer/
?   ??? SharedApiModels/
??? Tests/
    ??? ApiClientTests/
```

Each target is self-contained — just like in the original ExtremePackaging — but now with full OpenAPI integration, client/server symmetry, and end-to-end testability.

? Key Improvements Over ExtremePackaging

Area	Before	After
API Definition	Manual protocol layer	Generated OpenAPI spec
Networking	Custom client	Actor-based client w/ middlewares
Server	None	Vapor mock server (17 endpoints)
Authentication	Static token	BearerTokenAuthMiddleware
Logging	Simple print logs	Structured OpenAPILoggingMiddleware
Testing	Minimal unit tests	Full integration tests (54 total)
Schema Management	Handwritten	Modular YAML + YamlMerger
Tooling	Swift only	Swift + OpenAPI toolchain

? Lessons Learned

1. **OpenAPI fits perfectly into modular Swift architectures** — generated code belongs in its own layer, and SwiftPM makes that separation effortless.
2. **Actors are the future of shared state** — simple, safe, and transparent.
3. **Middleware > Managers** — function composition scales better than class hierarchies.
4. **Automation beats documentation** — with OpenAPI, the spec *is* the documentation.

? Closing Thoughts

This evolution of ExtremePackaging into a full OpenAPI reference app is more than a demo — it's a **blueprint for modular API-driven development in Swift**.

From YAML schemas to live servers and typed clients, everything now exists in one unified, testable ecosystem — powered by Apple's official OpenAPI tools and guided by the ExtremePackaging philosophy.

? Explore the project: swift-openapi-extremepackaging-example

---

"Architecture should scale not by adding layers, but by removing assumptions."