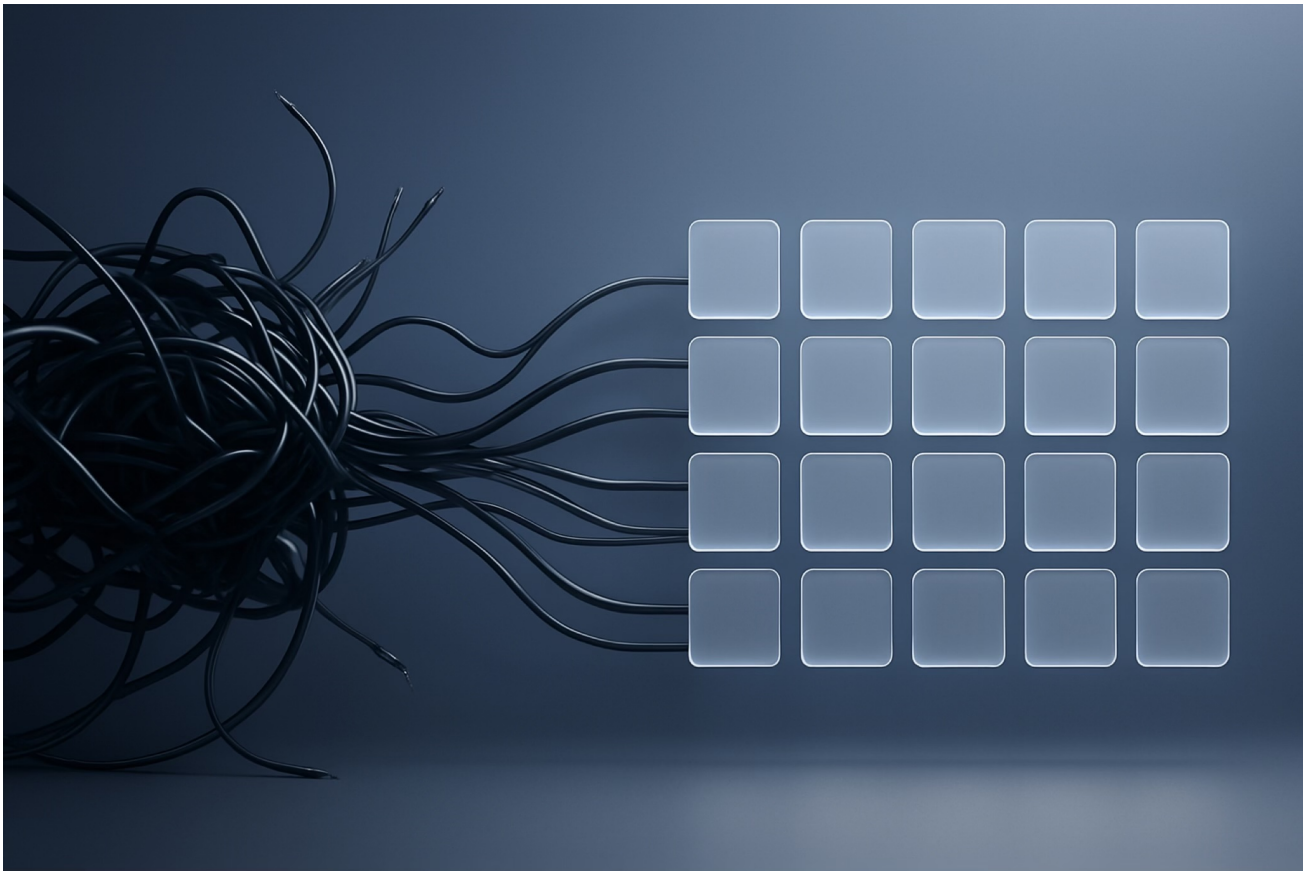
"Validations Are Values"



Validations Are Values

Here is the validation function almost everyone writes the first time. You have a parsed model. You want to check it. So you write this:

```
func validate(_ document: Document) throws {
    var errors: [String] = []
    if document.servers.isEmpty {
        errors.append("document has no servers")
    }
    for (path, item) in document.paths {
        for (method, operation) in item.operations {
            if operation.responses.isEmpty {
                errors.append("\(method) \(path) has no responses")
            }
            for variable in operation.server?.urlTemplate.variables ?? [] {
                if !operation.server!.variables.keys.contains(variable) {
                    errors.append("\(path) uses undefined variable \(variable)")
                }
            }
        }
    }
    // ... three hundred more lines ...
```

```
    if !errors.isEmpty { throw ValidationError(errors) }
}
```

It works. It also rots. Every new rule is another branch wedged into the same traversal. The error strings drift out of sync with the checks. Nobody can test a single rule in isolation, because there is no single rule, there is only the function. You cannot turn one check off without commenting out code. And the location of each failure is whatever you remembered to interpolate into the string, which is to say, inconsistent and usually wrong.

I stopped writing validators like this when I read Matt Polzin's OpenAPIKit. His validation layer is the cleanest I have seen in Swift, and the core idea is one sentence:

A validation is not code you run. It is a value you build.

Once you take that seriously, everything else follows. This post is the whole idiom, distilled from how OpenAPIKit actually does it. I now hold every validation layer I write to it.

What OpenAPIKit is

OpenAPIKit is a Swift library for reading, writing, and validating OpenAPI documents. OpenAPI is the specification format that describes an HTTP API: its paths, operations, request and response bodies, schemas, servers, and components. The documents are usually written as large YAML or JSON files.

OpenAPIKit gives you three things, layered cleanly on top of each other:

1. **A typed model of the entire OpenAPI specification.** Every construct in the spec, `OpenAPI.Document`, `OpenAPI.PathItem`, `Operation`, `Response`, `Server`, `JSONSchema`, and the rest, is a real Swift type. Not a `[String: Any]` you index into and pray. A real type with real fields, so the compiler knows the shape of an OpenAPI document.
2. `Codable` **conformance over that model.** Because the model is `Codable`, you decode a YAML or JSON spec straight into `OpenAPI.Document` and encode it back out. The decoder is where the first wall of correctness lives: a document that is not even shaped like OpenAPI fails to decode at all. This is the "parse first" pass.
3. **A validation subsystem over the decoded model.** This is the part this post is about. Once you have a decoded `OpenAPI.Document`, you run a `Validator` over it to catch the semantic errors that decoding alone cannot: a `$ref` that points at a component that does not exist, an operation with no responses, a server URL template that uses a variable nobody defined. These are valid OpenAPI *syntax* describing an invalid *document*.

The validation code lives under `Sources/OpenAPIKit/Validator/`. The files worth reading are `Validation.swift` (the atom), `Validator.swift` (the engine and the default set), `Validation+Builtins.swift` (the shipped rules), and `ReferenceValidations.swift` (the reference-resolution rules). The tests under `Tests/OpenAPIKitTests/Validator/` are as instructive as the source, and I will come back to them.

How you actually use it is three lines:

```
import OpenAPIKit
import Yams

let document = try YAMLDecoder().decode(OpenAPI.Document.self, from: yamlString)
try document.validate()   // throws a ValidationErrorCollection if anything is
wrong
```

The first line is the parse pass. The second is the validate pass. Everything below is how that

second line is built, and why it is built the way it is.

Parse first, validate second

The first move happens before any validation runs at all. Parsing and validation are two passes, not one.

The parser's job is to turn bytes into a well typed value and to make as many illegal states as it can simply unrepresentable. An enum cannot hold a case that does not exist. A non-optional field cannot be missing. By the time you have a parsed `Document`, a whole category of "errors" is gone, because the type system refused to let them exist.

Validation runs *after* that, over the already typed value. It catches only what the types cannot or should not enforce: a server template that references a variable nobody defined, an operation with zero responses, a reference that points at a component that is not in the document. These are not type errors. They are semantic ones. They need their own pass.

If you find yourself validating inside the parser, stop. Parse to a clean value. Then validate the value.

A validation is a small struct of closures

Here is the atom. A `Validation` over some subject type carries three things:

- a **description** string,
- a **check** that produces errors,
- a **predicate** that decides whether this rule even applies here.

That is it. No inheritance, no protocol with ten requirements. A plain value you can build, store in an array, pass around, and combine.

The simplest one reads almost like English:

```
public static var operationsContainResponses: Validation<Operation> {
    .init(
        description: "Operations contain at least one response",
        check: \.responses.count > 0
    )
}
```

Look at what is not there. There is no traversal. This rule does not know how to find `Operation` values in a document, and it does not care. It only knows what a correct `Operation` looks like. The machinery finds every operation in the tree and offers each one to this rule. Dispatch is directed by the subject type: a `Validation<Operation>` fires on operations, wherever they appear, and is invisible to everything else.

The description states the correct state, never the failure

Read that description again: "Operations contain at least one response." Positive. It describes the world when the rule is satisfied, not the world when it breaks.

This is deliberate, and it is one of my favorite details. You write the description once, phrased positively. When the rule fails, the framework derives the failure message by prefixing `"Failed to satisfy: "`. So you get:

> Failed to satisfy: Operations contain at least one response

One source of truth, two readings. You never write the failure phrasing by hand, so the check and its message can never drift apart. Compare that to the if-tree above, where `if responses.isEmpty` and the string `"has no responses"` are two independent things you have to keep in agreement forever.

The description does more than format messages. It is the rule's **identity**. Because rules are removed by matching their description, the description must be unique and stable. Which leads to the next piece.

Every error knows where it lives

An error is a reason plus a location:

```
ValidationError(reason: "Server Object does not define the variable 'x'", at:
context.codingPath)
```

You supply the reason. You do *not* supply the location by hand. The traversal fills in the coding path as it walks, so by the time your check runs, `context.codingPath` already says exactly where in the tree you are. The rendered message comes out consistent every time:

> ... at path: .paths./hello/world.get.servers

or, at the top:

> ... at root of document

No more interpolating `\(path).\(method)` into strings and getting it subtly wrong. The path is bookkeeping the framework owns, not you.

Application is predicate gated

Every validation has a `when:` predicate that runs before the check, with the full context in hand. The default is "always." But when a rule should only apply to a subset, you scope it with the predicate, not with an `if` inside the check body.

This matters because it keeps the check pure. The check answers "is this value correct," full stop. The predicate answers "does this rule even apply here," separately. A rule that only applies to OpenAPI 3.1 documents, or only to operations at a certain path, or only when some other field is present, says so in `when:` and keeps its check clean. Conditionals do not leak into the logic that decides pass or fail.

Two shapes, by how a value can fail

There are two ways to author the check, and the choice is mechanical.

If a value can fail **one way**, use the Bool form. You give a positive description and a predicate; `false` yields exactly one error, auto-formatted from the description. That `operationsContainResponses` rule above is this form. It is the dominant case.

If a single value can fail in **several places at once**, use the multi-error form and return one `ValidationError` per offender:

```
public static var serverVariablesAreDefined: Validation<Server> {
    .init(
        description: "All server template variables are defined",
        check: { context in
            context.subject.urlTemplate.variables
                .filter { !context.subject.variables.contains(key: $0) }
                .map { name in
                    ValidationError(
                        reason: "Server Object does not define the variable
'\(name)'",
                        at: context.codingPath
                    )
                }
        }
    )
}
```

A server with three undefined variables produces three errors, each with its own reason and its own path. You do not collapse them into one "server is invalid" message, because then you would lose two thirds of the information about what to fix. The rule emits one error per problem.

A default set you can add to and subtract from

You ship two validators: a populated one with a curated default set, and a `blank` one with nothing. Builders are fluent and chainable. You compose your actual policy from the defaults, plus your own rules, minus the ones you do not want:

```
let validator = Validator()                        // the defaults
    .validating(.operationsContainResponses)      // add a builtin by name
    .validating(myCustomRule)                      // add your own
    .withoutValidating("All server template variables are defined")  // remove
one by its description
```

This is why descriptions have to be unique and stable: the description *is* the removal key. `withoutValidating` matches on it. Rename a rule's description and you have quietly changed its identity.

When strictness comes in grades, you model the grades as whole-set swaps on the validator, not as flags threaded through individual rules. OpenAPIKit does lenient reference checks by default, swaps in the strict set with `validatingAllReferencesFoundInComponents()`, and clears them with `skippingReferenceValidations()`. Three named configurations, not a `strict: Bool` smuggled into every check.

The combinator algebra

Because validations and their checks are values, you combine them with an algebra instead of nesting `if` statements. These are the moves I reach for:

- `&&` **/** `||` combine checks. `&&` concatenates both error lists; `||` short-circuits and only fails if both sides fail.
- **comparison operators** (`==`, `>`, `>=`, and friends) lift a key path and a literal into a predicate, so `check: \.responses.count > 0` is a real expression, not a closure you wrote by hand.
- `take(\.path) { value in ... }` digs to a value and runs arbitrary logic past simple equality.
- `lift(\.child, into: ...)` runs child-typed validations against a child value while keeping the parent's path and document. This is the key compositional move for nested structure.
- `unwrap(\.optional, into:description:)` unwraps an optional, erroring if nil, otherwise running child validations on the unwrapped value.
- `lookup` **/** `unwrapAndLookup` resolve a reference against the document's component store, error if it is not found, otherwise validate the resolved value.
- `all(validations...)` applies many validations to the same context.

So a compound rule reads declaratively:

```
check: \.responseOutcomes.count >= 1
    && { $0.subject.responseOutcomes.allSatisfy { $0.status == 200 } }
```

You are describing what correct looks like by composing pieces, not writing a traversal.

Test the rules, the machinery, and the set

The test style mirrors the philosophy, and OpenAPIKit's bar here is the part most people skip. There are three layers to it.

Every rule gets a failing and a succeeding test. Not just "here is a broken document, does it throw." Also "here is a document that *almost* trips the rule but does not." The succeeding fixture is a near miss, the value sitting exactly at the boundary the rule guards. A rule tested only in its failing direction is untested, because nothing proves it stays quiet on valid input. Assert the thrown collection's exact count, each reason (including the `"Failed to satisfy: "` prefix), and each coding path:

```
func test_serverCountCheckFails() {
    let document = // ... seed the failing case
    let validator = Validator.blank
        .validating("All server arrays have more than 1 server", check:
\[Server].count > 1)
    XCTAssertThrowsError(try document.validate(using: validator)) { error in
        let error = error as? ValidationErrorCollection
        XCTAssertEqual(error?.values.count, 1)
        XCTAssertEqual(error?.values.first?.reason,
                       "Failed to satisfy: All server arrays have more than 1
server")
        XCTAssertEqual(error?.values.first?.codingPath.map(\.stringValue),
                       ["paths", "/hello/world", "get", "servers"])
    }
}
```

The machinery gets its own negative tests. The type-erased wrapper that filters by runtime type has to be proven correct on its own: an optional subject yields no errors, a wrong type yields no errors, a false predicate yields no errors. And one positive control, the same value of the same type appearing twice yields two errors, to prove those negatives are not vacuously passing.

The configuration is pinned. A test asserts the exact, ordered list of active rule descriptions for every validator configuration. `blank` is empty. The full default set is pinned description by description. This means you cannot add, remove, or reword a rule without a test failing first, which forces every such change to be deliberate rather than accidental.

The point of all this is that a validator can pass every test it ships and still be incomplete, if some field of the input model never got a rule at all. So the final discipline is a coverage law: keep a registry of every field the model can carry, and a meta-test that fails if any field lacks either a validation or an explicit "unsupported, and here is why" classification. Silent gaps get caught mechanically, not in review.

Why I hold everything to this now

I do not treat this as an OpenAPI trick. I treat it as the standard for any validation layer over any parsed model: a config, a manifest, an HTML tree, a schema, a feature model. One framework, parameterized over both the subject and the document, reused by every layer instead of copied per layer. Each rule a named, removable, isolation-testable value with a positive description. Errors that carry their own location. Combinators instead of conditionals.

The if-tree at the top of this post produces correct results. That is not the bar. The bar is whether you can add a rule without touching a three-hundred-line function, turn one off by name, test it alone, and trust that its failure message points at the right place. The if-tree fails all four. Validations as values pass all four, and they do it because the design made the right thing the easy thing.

Parse first. Then build your validations as values.

Credit where it is due: this entire idiom is Matt Polzin's. Go read `Sources/OpenAPIKit/Validator/`. It is some of the most quietly excellent Swift in the ecosystem.