JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#44

MASTERING MIDDLEWARE & ROUTING CHAINS

ARCHITECTING CLEAN, REUSABLE LOGIC FOR EVERY REQUEST

The Middleware Pattern

Handlers Wrapping Handlers

In Go, middleware is simply a function that takes an *http.Handler* and returns a new *http.Handler*. This allows you to execute code before the main logic (e.g., checking an API key) and after the logic (e.g., logging the response time).

The Philosophy:
Instead of bloating your business handlers with "boilerplate" code, you move that logic into reusable wrappers. Because everything satisfies the *http.Handler* interface, these wrappers can be chained together indefinitely.

Implementing Basic Middleware

The Request Logger

A common entry-level middleware is a logger. It records the method, path, and duration of every request. By using the *HandlerFunc* adapter, we can wrap any existing handler with this logging capability.

```go
func LoggingMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()

        // Pass control to the next handler in the chain
        next.ServeHTTP(w, r)

        log.Printf("%s %s %v", r.Method, r.URL.Path, time.Since(start))
    })
}
```

The Middleware Chain

Building the Pipeline

As your application grows, you might have five or ten middleware layers. Manually nesting them - *A(B(C(Handler)))* - quickly becomes unreadable and error-prone.

The Strategy:
Use a "Constructor" pattern or a lightweight helper to flatten the chain. This makes the order of execution explicit: requests flow through the middleware in the order they are defined, hitting the "outer" layers first and the "inner" business logic last.

```go
// A simple way to chain multiple middlewares
func Chain(h http.Handler, middlewares ...func(http.Handler) http.Handler) http.Handler {
    for _, m := range middlewares {
        h = m(h)
    }
    return h
}
```

Advanced Routing Patterns

Dynamic Paths and Methods

The standard *ServeMux* is powerful but basic. As of Go 1.22, it supports HTTP methods (GET, POST) and path wildcards (e.g., */users/{id}*) directly in the pattern string.

The Concept:
Seniors know that complex routing often doesn't require a heavy framework. By leveraging the new *ServeMux* features, you can keep your dependencies to zero while still handling RESTful resources and variable path segments elegantly.

```
mux := http.NewServeMux()

// Pattern matching with methods and wildcards (Go 1.22+)
mux.HandleFunc("GET /users/{id}", func(w http.ResponseWriter, r *http.Request) {
    id := r.PathValue("id")
    fmt.Fprintf(w, "Fetching user %s", id)
})
```

Organizing API Versions

Sub-Routing with Mux Nesting

When building large APIs, you often need to group routes by version (e.g., */api/v1*) or by resource. You can nest *ServeMux* instances to create a clean, tree-like structure for your application.

The Strategy:
Create a "Main Mux" that delegates specific prefixes to "Sub-Muxes". This keeps your routing table modular and prevents a single *main.go* file from becoming a thousand-line routing disaster.

```
v1 := http.NewServeMux()
v1.HandleFunc("/users", listUsers)

mainMux := http.NewServeMux()
// Strip the prefix before passing to the sub-mux
mainMux.Handle("/api/v1/", http.StripPrefix("/api/v1", v1))
```

Passing Data via Context

Communication Between Layers

Often, a middleware (like Auth) needs to pass information (like a UserID) to the final handler. Since we cannot change the *ServeHTTP* signature, we use the request's *Context*.

Best Practice:

Treat the context as a secure transport for request-scoped data. Extract the information in your middleware, "inject" it into a new context, and pass it down. This keeps your business handlers decoupled from the specifics of how the user was authenticated.

```
// Inside Auth Middleware
ctx := context.WithValue(r.Context(), userKey, userID)
next.ServeHTTP(w, r.WithContext(ctx))
```

Error Handling in Chains

Centralizing Failure

One of the greatest benefits of middleware is centralized error and panic handling. You can create a "Recovery" middleware at the very top of your chain to catch panics, log the stack trace, and return a clean 500 error to the user.

The Logic:
By wrapping your entire server in a recovery layer, you prevent a single "nil pointer" in one handler from crashing the entire process. It ensures the server stays up and the client receives a valid HTTP response even when things go wrong internally.

Summary:

- **Middleware**: Use the decorator pattern to isolate cross-cutting concerns.
- **Modern Routing**: Use Go 1.22+ path patterns for RESTful APIs.
- **Modularity**: Nest *ServeMux* instances to organize large APIs.
- **Context**: Use it to safely share metadata across the middleware chain.

Next, we flip the script and look at the other side of the wire: Building Robust HTTP Clients with Timeouts and Retries.