Go Backend

For Developers Who Actually Want To Understand It

A backend guide
from first principles

Diwash Pandey

What we will be learning

Read This First (I Mean It)

Honestly, I actually wrote this book for myself. Sounds selfish, I know.

When I was learning Go backend, I couldn't find anything that taught it the way I wanted to learn free, everything in one place, **and mainly from complete fundamentals**. So I just started writing my own notes. The way I always do. Deep, first principles, no skipping the "why".
That's just how I learn. I can't just use something without knowing how it actually works underneath.

I wasn't planning to share it. It was just... my notes.

Then one day I read it back and thought... wait, this is actually good. If this is helping me understand things, it'll help others too. So I just decided to make it public. That's it. No big plan. No marketing strategy. Just a guy who wrote something useful for himself and thought "why not share it with others"

And since I didn't write it to sell it, it's free, Completely.

Scan the following QR to get to the official site.

learngobackend.com

I genuinely want it accessible to everyone so I'm publishing it for free.

Fair warning though → this is not a professional book. Don't expect fancy writing or a suit and tie. Expect real explanations, first principles thinking, and a writing style that feels like your senior dev friend explaining things over coffee. It's a living document. It'll keep getting better.

Who this Book is For

You already know backend. Maybe you've been writing in Node, Python, Java, or whatever... and then you found Go. And something clicked. The simplicity, The Speed, The way it just makes sense. Now you want to build a backend in Go, but you don't want to just memorize functions and move on. You want to actually understand what's happening.

That's exactly who this book is for.

This book is for backend developers from other languages who want to migrate to Go... not by copy-pasting from Stack Overflow or ChatGPT, But by understanding Go from the ground up. We go deep. We start from how things actually work under the hood, then build up to how to use them.

By the end, you won't just know how to write a Go backend - **You'll know how to learn Go.** Which honestly is worth more than any single topic I could teach you.

And yeah, that's a bold thing to say But I mean it.

What You Must Already Know

This book assumes you're not starting from zero.

You should already know backend concepts at an intermediate level. HTTP, requests, responses, databases, authentication. You don't need to be an expert, but you should have built something real in some language before.

You also need to know the Go language itself, variables, structs, interfaces, goroutines, the basics. This book does not teach you Go syntax. It teaches you Go backend.

One more thing: you need the habit of learning things deeper instead of memorizing them. If you're someone who reads a code example and asks "okay but why does that work?"... this book is written for you. If you just want to copy-paste working code and ship it, this book will frustrate you. And that's okay. Different books for different people.

What This Book Is Not

This is not a professional project book. You won't build a full application by the end of it. What you will have is a solid understanding of how every moving part works, so when you do build something, you'll know exactly what you're doing and why.

This book is also at an early stage. It's v0.1. That means a few production-level topics are still missing like OAuth authentication, deployment, a project, and a few other things. I'll be honest about that upfront rather than pretend otherwise.

But here's what I want you to understand: missing a few topics doesn't mean this book leaves you incomplete. It covers 80%+ of what you'll actually use in production... logging, context injection, database integration, JWT auth, testing, middleware... **all from first principles.** Even the things this book doesn't cover yet, you'll be able to learn easily on your own. **Because you'll have the mental model. You'll know how Go thinks.**

The missing topics? They're coming in future versions. Maybe before you even finish reading this one.

Have I Used AI In This Book?

Of course I have. Who doesn't these days? ;)

But let me be clear about what that means. I didn't ask AI to write this book for me. I wrote every concept, every explanation, every mental model myself... while actually learning these things, building things, breaking things. AI helped me catch mistakes and polish my English. That's it.

Also, let's be real, you might be using AI to summarize this book right now. Who knows. I'm not judging. We're all using the tools available to us. The question is whether the thinking behind it is real. Mine is.

Fun Fact: I wrote this book while learning Go backend myself. haha.

How to Read This Book

A few things that'll make this 10x more effective:

1. Let go of your old backend structure :
Bring your backend concepts with you... HTTP, routing, databases, auth. But forget the framework-specific way your old language does things. Go has its own way. Trust it.

2. Follow first principles :
Every time I explain something, I start from how it actually works under the hood — before showing you how to use it. Don't skip those parts. That's where the real learning happens. I promise it's worth it.

3. Break things :
I'll show you how things are built. You go and write the code, then mess with it. Remove a line. Change a value. Try to make it crash. Try to make it do something it's not supposed to do. That's how you actually learn. Just don't break your laptop.

4. Don't memorize — understand :
If you're reading a section and you find yourself trying to memorize the function signatures, stop. Go back and ask: why does this exist? What problem is it solving? The memorization will happen naturally once the understanding is there.

5. Use the cross-references.
This book connects concepts across sections intentionally. When I say "check section X," actually go check it. The connections are the point(like the dots you connect)

Versions Used

Go Version Used

This book was written using Go 1.25.

Go has a strong compatibility promise. Almost everything written for Go 1.x works across versions without breaking changes. So even if you're reading this a year or two after it was written, the code here will almost certainly work fine. Go takes backward compatibility seriously. It's one of the best things about the language.

If something does break, the Go release notes will tell you exactly what changed and why. But genuinely, don't worry about it.

Package Versions Used

Package	Version	Install
github.com/rs/zerolog	v1	go get github.com/rs/zerolog
github.com/golang-jwt/jwt/v5	v5	go get github.com/golang-jwt/jwt/v5
github.com/jackc/pgx/v5	v5	go get github.com/jackc/pgx/v5
github.com/pkg/errors	v0.9.1	go get github.com/pkg/errors

⇒ All other packages used in this book (net/http, encoding/json, context, testing, net/http/httptest) are part of Go's standard library. No installation needed.

NOTE : Even if you're reading this months or years after publication, these versions will almost certainly still work. None of these packages have a history of frequent breaking changes. Go's ecosystem moves deliberately, not chaotically. But if you hit something weird, always check the package's GitHub releases page — the changelog will tell you exactly what moved and why.

Let’s start and break things, smartly.

1. Foundation

⇒ In Go, you don't need any external framework to build a backend.
⇒ Because:
→Go's standard library already includes a production-grade HTTP server
→ That server lives in **`net/http`** package

1.1. Introduction to `net/http`

⇒ `net/http` is a package available in Go's standard library

⇒ It abstracts the raw TCP connections and HTTP protocol:
→ Parsing HTTP requests
→ Request Routing
→ Managing headers and responses

⇒ What Go saying is:
→ Here is a strong foundation. You build logic on top of it

1.2. First Principle View of a Web Server

(You control behavior, Go controls protocol)

⇒ Let's forget Go for a moment

⇒ At the lowest level the internet runs on TCP
⇒ **TCP** → is just stream of bytes
⇒ **HTTP** → is just a set of rules about how TCP bytes should be structured

⇒ A web server is simply just a program that:
→ Listen for TCP connections
→ Reads incoming bytes
→ Interprets them as HTTP
→ Runs some logic
→ Writes response bytes back

1.3. How a Go Web Server Works

Step 1: Opens a TCP connection port
Step 2: Accept incoming connections from clients
Step 3: Read incoming bytes from the connection
Step 4: Parse those bytes into an HTTP request(method, headers, body, etc)
Step 5: Decide which code to run based on the URL (routing)
Step 6: Run that code (business logic that prepares the response)
Step 7: Write HTTP response bytes back to the connection.

1.4. How You And Go Work Together

⇒ We've seen how GO web server works in previous section
⇒ Now let's see how You and Go do it together

Step 1 :

→ **You:** Just configure the settings
→ **Go :** Starts and manages TCP connections

Step 2, Step 3, Step 4 :

→ **Go does everything**
→ (accept connection, read bytes, parse HTTP request)

Step 5:

→ **You:** Setup the routes (router/mux)
→ **Go:** Runs Codes based on your routing when the request arrive

Step 6:

→ **Only You**
→ write business logic and prepare HTTP response
→ Return the HTTP Response to Go

Step 7:

→ **Go**
→ Converts response to bytes
→ Send it back to the connection

Go handles networking and HTTP
You handle logic and decisions

2. Core Server Setup

2.1. Introduction: The 3 Pillars of Go Backend

⇒ Every Go backend revolves around **3 main things.**
⇒ And if you understand these **3 things** you've understood **big part of Go Backend**:

1. Server
The Server manages incoming and outgoing HTTP requests

2. Router
The Router decides which handler should run for a request

3. Handler
The Handler runs your actual business logic

Example: Basic Server Code Example

```
func main() {
                                    * - - - - - - Router - - - - - - *
    mux := http.NewServeMux()
    mux.HandleFunc("/", handler1)          <- - - - - Registering Handler to router
    mux.HandleFunc("/hello", handler2)     <- - - - - Registering Handler to router

                                    *- - - - - - Starting Server - - - - - - *
    err := http.ListenAndServe(":8080", mux)

    if err != nil {...}
}

                                    * - - - - - - Handlers - - - - - - *
func handler1(w http.ResponseWriter, r *http.Request) {
    var response []byte = []byte("Hi from Base !!")
    w.Write(response)
}
func handler2(w http.ResponseWriter, r *http.Request) {
    var response []byte = []byte("Hello there !!")
    w.Write(response)
}
```

- - - ⇒ Didn't understand right? Don't worry we're just starting"" ⇐ - - -

2.2. Server

⇒ Now we start unpacking things one by one
⇒ We'll begin with the Server, because it's the entry point

2.2.1. Introduction to `http.Server`

⇒ `net/http` provides us `http.Server` to configure and run the server.
⇒ `http.Server` is like a engine that runs your server
⇒ Manages things like:
→ Running HTTP server
→ Listening on a network address
→ Accepting incoming requests
→ Passing those requests to handler

2.2.2. Core Structure of `http.Server` [For Beginner]

BIG NOTE

⇒ `http.Server` has many fields and features
⇒ Right now, we'll look at important basics only.

⇒ Later in the book, when things make more sense,
⇒ we'll come back and learn the advanced parts.

```
type Server struct {
        Addr                    string
        Handler                 Handler
        ReadTimeout             time.Duration
        ...
}
                        * - - - - - - - -  Important Methods  - - - - - - - - *
func (s *Server) ListenAndServe() error
func (s *Server) ListenAndServeTLS(certFile, keyFile string) error

                        * - - - - - - - -  Helper Functions (Shortcuts)  - - - - - - - - *
func ListenAndServe(addr string, handler Handler) error
func ListenAndServeTLS(addr, certFile, keyFile string, handler Handler) error
```

- - - - - ⇒ Now let's understand what they do, in the next page ⇐ - - - - -

type Server struct {...} :

⇒ The main Go HTTP server structure
⇒ It holds all server-level configuration
⇒ You control how your server behaves by setting its fields
⇒ After configuration, you start it using **`server.ListenAndServe()`**

* - - - - - - - - - - - Fields - - - - - - - - - - - *

Server.Addr :

⇒ The network address where your server will run
⇒ Eg: → **":8080"**
→ **"127.0.0.1:8080"**
⇒ When you start the server, Go listens for requests on this address.

Server.Handler :

⇒ Your Server will pass each request to this Handler
⇒ You usually put a router (ServeMux) here
⇒ **How it works**
→ Server receives a request
→ Server parse it as **`http.Request`** (we'll learn about it)
→ Server passes request it to this handler **`Server.Handler`**
⇒ So you give router to this field

Server.ReadTimeout :

⇒ Maximum duration to read the entire request, including body.
⇒ Protects against slow clients

* - - - - - - - - - - - Method - - - - - - - - - - - *

Server.ListenAndServe() error:

⇒ Start the HTTP server on address **`Server.Addr`**

* - - - - - - - - - - - Helper Function - - - - - - - - - - - *

ListenAndServe(portString, mux) error:

⇒ Use when you don't want to configure **`http.Server`**
⇒ **Internally:** → creates a default `http.Server`
→ sets the address
→ sets the handler (mux)
→ starts the server

2.2.3. [Example] : Configuring Basic Server and Running it

```
func main() {
                                * - - - - Router + Handler - - - - *
                        *- - - - (we'll learn router and handler later) - - - - *
        mux := http.NewServeMux()
        mux.HandleFunc("/", OurHandler)

                                * - - - - - - - Configuring Server - - - - - - - *
        server := &http.Server{
            Addr:       ":8080",          //  Server will run in this port
            Handler:    mux,              // Server will forward request to handler (router)
            ReadTimeout:  5 * time.Second,      // Extra configuration
        }

                                * - - - - - - - Starting the Server - - - - - - - *
        err := server.ListenAndServe()
        if err != nil {
                // handle error
        }
}

                        * - - - - - - - The Handler (we'll learn later) - - - - - - - *
func OurHandler(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello from custom server"))
}
```

What's Happening Here ?	How It Works
⇒ We created a router ⇒ We registered a handler inside the router (we'll learn both in detail soon) ⇒ We created an `http.Server` ⇒ Configured it with: → address → router as handler → basic timeout	⇒ When the request arrives ⇒ `http.Server` receives raw bytes ⇒ Converts bytes into `http.Request` ⇒ Passes the request to its handler → the router that we gave it ⇒ Router checks the route ⇒ Matching handler runs

⇒ We started server with `ListenAndServe()`	

2.2.4. [Example] : Running Server with Helper `ListenAndServe()`

```
func main() {
                        * - - - - Router + Handler (we'll learn later) - - - - *
        mux := http.NewServeMux()
        mux.HandleFunc("/", OurHandler)

                        * - - - - - - - Starting Server Directly - - - - - - - *
        err := http.ListenAndServe(":8080", mux)
        if err != nil {
                // handle error
        }
}

                        * - - - - - - - The Handler (we'll learn later) - - - - - - - *
func OurHandler(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello from custom server"))
}
```

What's Happening Here ?

⇒ We created a router
⇒ We registered a handler inside the router
(we'll learn both in detail soon)

⇒ We directly started server with
→ http.ListenAndServe(address, handler)

⇒ `ListenAndServe()` Internally:
→ creates a default `http.Server`
→ sets the address
→ sets the handler (mux)
→ starts the server

2.3. Router

⇒ Routing means deciding which code handles a request, based on
→ HTTP Methods
→ Host (optional)
→ and mainly: URL path.

⇒ In the previous section, we passed something called `mux` to the server's handler
⇒ So, what exactly is `mux`

2.3.1. Introduction to `http.ServeMux`

⇒ `http.ServeMux` is router provided by `net/http`
⇒ It's job is simple:
→ A request comes in
→ `http.ServeMux` checks the request path
→ and, decides which handler should run

2.3.2. Core Structure of `http.ServeMux`

```
type ServeMux struct {
        ...             - - - - - - > Internal fields to store handlers and routes
}

                        * - - - - - - Creation Function  - - - - - - *
func NewServeMux() *ServeMux

                                * - - - - - - Methods  - - - - - - *
func (mux *ServeMux) Handle(pattern string, handler Handler)
func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter,
*Request))

func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request)
        ↘
        You don't directly use ServeHTTP()
                → but it's here for some important information
```

type ServeMux struct {} :

⇒ `net/http` use this struct as router
⇒ Stores mapping of URL patterns to handlers internally
⇒ Matches incoming requests to the matched URL

ServeMux.Handle(pattern, handler) :
⇒ Registers a handler for a given URL pattern

ServeMux.HandleFunc(pattern, handlerFunction) :
⇒ Converts passed handler function into HandlerFunc to satisfy Handler interface
⇒ Registers that Handler for a given URL pattern

NewServeMux() *ServeMux :
⇒ Creates and returns a new instance of ServeMux (router)

ServeMux.ServeHttp(w, r) :
⇒ You don't directly use this method
⇒ But I've included it so we can understand deep how ServeMux works
⇒ We'll talk about it in the coming section
→ `4.6. [IMPORTANT] : Deep Dive into http.ServeMux and http.Server`

2.3.3. [Example]: Using ServeMux as Router

```
mux := http.NewServeMux()
mux.HandleFunc("/hello", handler)

http.ListenAndServe(":8080", mux)

func main() {
                    * - - - - Router + Handler (we'll learn later) - - - - *
        mux := http.NewServeMux()
        mux.HandleFunc("/hello", OurHandler)

                        * - - - - - - - Starting the server - - - - - - - *
        err := http.ListenAndServe(":8080", mux)
        if err != nil {
                // handle error
        }
}

                    * - - - - - - - The Handler (we'll learn later) - - - - - - - *
func OurHandler(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello from custom server"))
}
```

Step 1: `NewServeMux()` created new `ServeMux`
Step 2: `HandleFunc()` registered **OurHandler** for **"/hello"** path
Step 3: Passed mux to `ListenAndServe()`
→ **`http.Server`** now will use this mux as its handler

2.3.4. Understanding Pattern Format

Syntax

[METHOD] [HOST] / [PATH]

Exact Match :

⇒ If your pattern doesn't end with "/"

→ Pattern must match the path exactly.

"/foo"

Only Matches ✅	Doesn't Match ❌
"/foo"	→ "/foo/other" → "/foo/"

Prefix match (Trailing Slash) :

⇒ If your pattern ends with "/"

→ Pattern matches anything that starts with it.

"/foo/"

Matches ✅	Doesn't Match ❌
"/foo/" **"/foo/abc/"** **"/foo/abc/def"**	**"/foo/other"** **"/foo/"**

Exact Root Match ("{$}") :

⇒ If your pattern ends with "/",

→ It matches anything that starts with it, right?

⇒ That means "/random", "/abc/xyz", everything falls through to "/".

⇒ If you want "/" to match only "/", use:

"/{$}"

⇒ Only matches with → "/"

METHOD Specific Patterns :

⇒ You can specify the HTTP Method directly in the pattern too.

"GET /foo"

Matches ✅	Doesn't Match ❌
GET request on **/foo**	**POST** request on /foo

HOST Specific Patterns :

⇒ You can specify the domain/host directly in the pattern too.

"example.com/foo"

⇒ Only matches request if

→ **Host == "example .com"** and path is **"/foo"**

Path Parameters (Wildcards) :

"GET /users/{id}"

⇒ We'll learn more about this soon...

Catch-All Parameters :

"GET /users/{path...}"

⇒ We'll learn more about this soon too...

2.3.5. [Important]: Pattern Matching Priority

⇒ When multiple patterns match a request path

⇒ The longest and most specific pattern wins.

For Eg:

- "/channel/images/thumbnails/"
- "/channel/"
- "/channel/images/"

⇒ If a request comes for "/channel/images/thumbnails/123.jpg"

→ All three patterns partially match.

→ but "/channel/images/thumbnails/" is the longest, so it handles the request.

2.3.6. [Important]: Deep Dive into http.ServeMux and http.Server

Read Me

⇒ If you are just starting this Book, you can safely skip this section for now
⇒ Come back after you understand Handlers & Middlewares

Mental Build up :

⇒ We already know :

→ If something implements ServeHTTP method,
→ Go can use it as a handler

```
type Handler interface{
   ServeHTTP(ResponseWriter,
*Request)
}
```

Now the important realization

⇒ We saw that `http.ServeMux` has method `ServeHTTP`

```
type ServeMux struct {
        ...
}
func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request)
```

⇒ Which means:

→ ServeMux itself is a handler
→ ServeMux is actually a handler which acts like a middle ware

⇓ - - - - - - **Now comes the real part** - - - - - - ⇓

How `http.Server` and `http.ServeMux` works together :

Step 1: Server receives raw HTTP Request

```
http.ListenAndServe(":8080", mux)
```

⇒ OS accepts TCP connection
⇒ `http.Server`:
→ Reads request
→ Parse it into structured `http.Request`
→ Creates `http.ResponseWriter`
⇒ THEN, `http.Server` calls:

```
mux.ServeHTTP(w, r)
```

Step 2: ServeMux acts like a middleware

⇒ Reads URL from `r.URL.Path`
⇒ Matches it to the **routes that we registered**

```
mux.HandleFunc("/hello", handler)
```

⇒ Finds the best matching handler
⇒ Calls that handler's `ServeHTTP(w, r)` ... exactly like a middleware

So, what we've understood :

⇒ `ServeMux` is a Handler that behaves like middleware by delegating to other handlers.
→ It does its job : Routing
→ Then forwards the request to another handler
→ Same pattern as logging, auth, and other middlewares

2.4. Handler

Until now, We've learned:

⇒ Server receives request and passes it to ServeMux(Router)

⇒ ServeMux chooses handler

Now the question is:

⇒ What exactly is a handler?

2.4.1. Introduction to Handlers

⇒ Handler is just a function that:

→ Handles an HTTP request

→ Writes back HTTP request

⇒ You mostly write Handler functions in your backend

⇒ Inside handler, you can:

→ Read request data

→ Run business logic

→ Write back response

Visual Flow

Client → Server → ServeMux → Handler

* The handler is the final destination of a request *

2.4.2. [Important]: The `http.Handler` Interface (The Core of Everything)

⇒ Go's HTTP system understands only ONE shape of handler

```
// http.Handler
type Handler interface{
   ServeHTTP(ResponseWriter, *Request)    // http.ResponseWriter & http.Request
}
```

⇒ If anything has `ServeHTTP` method
→ then Go can use it as handler

2.4.3. How Handlers Are Executed

⇒ Let's connect everything clearly

⇒ When a request comes:

Step 1: Server receives raw bytes of request
Step 2: Server parses raw request bytes into `http.Request`
Step 3: Server calls `ServeHTTP` of `ServeMux` that we gave
Step 4: `ServeMux` finds the handler matching with route that we registered
Step 5: `ServeMux` calls the `ServeHTTP` method of our handler

⇒ So in the end:

```
handler.ServeHTTP(responsewriter, request)
```

That's where your code runs

2.4.4. Understanding `http.ResponseWriter` & `http.Request`

```
func handleHello(w http.ResponseWriter, r *http.Request) {
        ...
}
```

`http.ResponseWriter` :

⇒ You use it in your handler to write the Response

`http.Request` :

→ You use it in your handler to read the incoming Request from client

2.4.5. Two Ways to Create and Register Handlers

⇒ In Go, you can create handlers in two ways
⇒ Important: both ways end up with ServeHTTP()

Struct Based Handler (Explict)

⇒ Here, we explicitly create a **struct** that implements `ServeHTTP`

```
type HelloHandler struct{}
func (h HelloHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    ...
}

// Registering in mux using `mux.Handle` function
mux.Handle("/hello", HelloHandler{})
```

Function Based Handlers (Shortcut)

⇒ Instead of creating a struct, we just write a function

```
func hello(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello"))
}

// Registering in mux using `mux.HandleFunc` function
mux.HandleFunc("/hello", hello)
```

How Does HandleFunc() Work?

⇒ Your function does not have ServeHTTP()
⇒ But HandleFunc() handles that for you
⇒ What HandleFunc() does internally:
→ Converts your function into HandlerFunc
→ HandlerFunc has ServeHTTP()
→ So it satisfies Handler
→ Then it registers it into the mux

Final Understanding

`**Handle()**` → you give a real `Handler`
`**HandleFunc()**` → It converts your function into a `Handler`

3. Request Lifecycle inside Go net/http

In this section we'll learn about the entire request lifecycle of a HTTP request inside your Go's server

3.1. What is the request lifecycle?

⇒ Request lifecycle is something like a step-by-step journey your HTTP request takes inside Go's server from the raw TCP connection to moments your handler sends a response.

3.2. Why is it important to understand

⇒ Understanding this flow is really really important because it shows how Go manages concurrency, parsing, routing, and response-writing under the hood.

3.3. The LifeCycle

3.3.1. TCP connection Established

⇒ Your Go server listens on port you've given
→ calls `net.Listen("tcp", "your_port")` internally
⇒ Your OS listens on the specific port
⇒ When client initiates a TCP handshake, a TCP connection is created
⇒ This connection is a **bi-directional byte stream**
→ Not HTTP yet, just raw data flow(in stream)

3.3.2. TCP connection Handled Concurrently

⇒ The http.Server accepts the connection via Accept()
⇒ Spawns a new goroutine per TCP connection (Mostly)
→HTTP/2 multiplexes streams on one connection
⇒ This goroutine reads requests and writes responses concurrently with others.

3.3.3. HTTP Request Bytes Read from TCP

⇒ The goroutine starts reading bytes from the connection using buffered I/O
⇒ Reads until it detects a complete HTTP requests header block
→ ending with **"\r\n\r\n"**
⇒ If a request has a body (based on headers like Content-Length or Transfer-Encoding)
→ body bytes remain unread at this point.
⇒ **Headers are fully parsed and stored. The body is left unread (lazy).**

3.3.4. HTTP Request Parsed into "http.Request"

⇒ The server parses raw bytes into an **`http.Request`** object
⇒ Parsing steps:
→ Extract from **request line**: method, path, protocol version
→ Parse **Headers** into a map (**`r.Header`**)
→ Parse **URL path** and **Query parameters** separately
→ **Body is exposed as an io.ReadCloser stream (r.Body) for lazy reading by handler.**
→ Attach a context.Context to Request which manages cancelation and deadlines

3.3.5. Request Routed by ServeMux or Custom

⇒ After parsing, Go calls **`Handler.ServeHTTP(ResponseWriter, *Request)`**
⇒ ServeMux matches the request against the patterns you registered.
⇒ If no match: 404 Not Found.
⇒ If method mismatch: 405 Method Not Allowed.
⇒ **NOTE:** This routing decision happens inside ServeMux, before your handler logic runs.

3.3.6. Handler Executes (Your major part in development)

⇒ Once routing chooses the handler:
→ It invokes its **ServeHTTP(**w, r**)** method of that handler.
⇒ Handler gets:
→ **`http.ResponseWriter`** to build the response.
→ **`http.*Request`** with **fully parsed headers** and **lazily-read body stream.**
⇒ Handler can:
→ Read headers anytime (**`Request.Header.Get`**).
→ Read body on demand via **`Request.Body.Read()`**, **`io.ReadAll()`**, **`json.Decoder`**, etc.
→ Write response headers/status/body via **`ResponseWriter`**
⇒ Use **`Request.Context()`** for cancellation or deadlines

3.3.7. Writing the Response & Header Management

⇒ Headers are not sent immediately when you set them with `w.Header().Set()`
→ Until first write
⇒ Headers are sent only once you:
→ write the first byte of body with `w.Write()`
→ or explicitly call `w.WriteHeader()`
⇒ If WriteHeader is never called
→ Go automatically sends response header
→ Go sends 200 OK by default when the body starts.
⇒ You cannot change headers after the body starts sending.
⇒ **Body is streamed as you write bytes**
⇒ If the handler panics, Go recovers, logs, and sends 500 Internal Server Error.

3.3.8. Connection Handling: Keep-Alive or Close

⇒ If the client supports keep-alive or HTTP/2, the connection stays open.
⇒ Server reuses connection for next requests.
⇒ Server closes connection on errors, timeouts, or client disconnect.

3.3.9. Request Context and Cleanup

⇒ Handler finishes
⇒ Each `*Request` has an associated `context.Context` accessible via `r.Context()`
⇒ Request context cancels (timeout, client disconnect)
⇒ Clean up resources in your handler by listening for context cancellation.
⇒ Any goroutines started with request context should listen for cancellation
⇒ Server cleans up internal resources, freeing memory and file descriptors

3.4. Header and Body Streaming

Request Header :

⇒ When you handler starts, Go's server has already:

→ Read all the Request Headers from the TCP stream

→ Parses them into a clean `map[string][]string` accessible via `Request.Header`

⇒ This means:

→ You can immediately inspect any header (eg: Content-Type, Authorization, etc)

→ Headers are fully available and stable (no waiting or partial data)

⇒ No streaming here; headers come as whole block and are done parsing before your handler runs

Request Body :

⇒ Unlike headers, the request body is not read upfront.

⇒ Go exposes the body as an **`io.ReadCloser`** stream in **`r.Body`**.

⇒ Which means:

→ The body data arrives over TCP gradually, not at once.

→ Your handler reads the body on demand by calling Read methods.

→ You control when and how much to read:

- Read fully at once with **`io.ReadAll(r.Body)`**
- Stream and process chunks progressively using buffered reads or **`json.Decoder`**

⇒ Important:

→ You must close **`r.Body`** if you do not fully read it, it free connection resources.

→ If you ignore reading or closing, connections may leak or hang.

4. HTTP Message Structure

⇒ Every request and response is sent as a structured message so both sides know what action is requested, what happened, and how to interpret the data.

4.1. Structure

⇒ Every HTTP message follows the exact structure with the same order.

START-LINE
HEADERS
EMPTY-LINE
[BODY]

4.2. Key Mental Model

Start-Line → What action is done
Headers → How to interpret the request or response
Empty line → boundary to separate Headers and Body
Body → Actual data / Payload

4.2.1. Start-Line

⇒ The start-line defines the core meaning of the request or response
⇒ It is always the **first line**.
⇒ Start-Line has different things in Request and Response

	Start-Line in Request	Start-Line in Response
Structure	METHOD PATH PROTOCOL	PROTOCOL STATUS-CODE STATUS-TEXT
Example	GET /users/42 HTTP/1.1	HTTP/1.1 200 OK

4.2.2. Headers

⇒ Headers are metadata lines that come after the start-line.
⇒ They describe how to interpret the request and response.
⇒ They are metadata, not the actual data.

⇒ Headers are continued until an EMPTY-LINE is reached.

(We'll go deep into headers in its dedicated section)

4.2.3. Empty-Line

⇒ The empty line is a mandatory separator between headers and body.

⇒ Marks the end of headers
⇒ Signals the start of the body
⇒ Exists because HTTP runs on a byte stream

4.2.4. Body

⇒ The body is the payload of the HTTP message and is optional
⇒ It contexts things like : JSON, HTML, FormData, files, etc

(We'll go deep into body in its dedicated section)

4.3. HTTP Structure Examples

HTTP Request Example
POST /posts HTTP/1.1 **← start-line** Host: example.com **← header** Content-Type: application/json **← header** Content-Length: 27 **← header** **← empty line** {"title":"Hello World"} **← body**

HTTP Response Example
HTTP/1.1 201 Created **← start-line** Content-Type: application/json **← header** Content-Length: 35 **← header** **← empty line** {"id":1,"title":"Hello World"} **← body**

5. http.Request Anatomy

5.1. Introduction

⇒ `http.Request` is Go's structured representation of a **parsed HTTP request.**
⇒ It holds all the details of client's request to a server
⇒ It is populated automatically by `net/http`

⇒ `http.Request` is used in two very different roles
1. Server-Side: represents an incoming request
2. Client-Side: represents an outgoing request

Mutability :

⇒ `http.Request` is **mutable**
⇒ But modifying it on the Server-Side doesn't make sense cause it's an incoming client request
⇒ You'll create and modify it on 2 scenarios:
1.Client-Side:
⇒ Used to create and send HTTP request to other APIs as Client
⇒ We'll learn about it in section '18. Go as Client'
2. Testing:
⇒ Used to create and modify fake requests to test handlers
⇒ We'll learn about it in section '19.2.3. How to create fake `Request'

When is it created :

⇒ It is created by `net/http` after:
→ TCP bytes are read
→ HTTP protocol is parsed
→ Headers, method, URL are validated
⇒ Everything's done behind the scenes
⇒ and Remember "body" is still not loaded (cause it's streamed)

5.2. Core Structure of `http.Request` (For Server-Side)

NOTE

⇒ We're learning the core structure of it for Server-Side

⇒ We'll look at it for Client-Side in section

→'18.4. Core Structure of `http.Request` (For Client-Side)'

```
type Request struct {
   Method       string          - - - - - - - - - - > contains HTTP methods (eg. GET, POST, etc)
   URL          *url.URL        - - - - - - - - - - > Parsed URL
   Proto         string          - - - - - - - - - - > HTTP protocol version(eg, HTTP/1.1)
   ProtoMajor    int             - - - - - - - - - - > HTTP protocol version major number(eg, 1)
   ProtoMinor    int             - - - - - - - - - - > HTTP protocol minor number(eg, 1)
   Header        Header          - - - - - - - - - - > HTTP request headers (map)
   Body          io.ReadCloser   - - - - - - - - - - > Request body stream
   ContentLength int64            - - - - - - - - - - > Body size if known
   TransferEncoding []string      - - - - - > Transfer encoding types (e.g., chunked)
   Host          string           - - - - - - > Host Header (domain name)
   Form          url.Values      - - - - - - - - - - > All parsed form data (query + POST)
   PostForm      url.Values      - - - - - - - - - - > Form data from POST/PUT/PATCH only
   MultipartForm *multipart.Form  - - - - - - - - > Parsed file uploads and form fields
   Trailer       Header          - - - - - - - - - - > Trailer headers sent after body
   RemoteAddr    string          - - - - - - - - - - > Client IP and port
   RequestURI    string          - - - - - - - - - - > Raw request URI (server side only)
   TLS           *tls.ConnectionState  - - - - - - - - - > TLS info if HTTPS connection
}
```

```
Context()                 - - - - - - - - - - > Returns requests `context.Context`
WithContext(ctx)          - - - - - - - - - - > Returns shallow copy with new context
BasicAuth()               - - - - - - - - - - > Extracts Basic Authentication username and
password
Referrer()                - - - - - - - - - - > Returns Refer header
UserAgent()               - - - - - - - - - - > Returns User-Agent form the header
ProtoAtLeast(major, minor)  - - - - - - - - - - > Checks if HTTP version >= specified
Write(w io.Writer)          - - - - - - - - - - > Writes raw HTTP request(not used by
you)
PathValue(name)             - - - - - - - - - - > Gets wildcard path parameter value
Clone()                   - - - - - - - - - - > Returns a deep copy of the request with a new
context
Cookies()                 - - - - - - - - - - > Returns all cookies sent by the client
CookiesNamed(name)          - - - - - - - - - - > Returns all cookies with the given name
Cookie(name)              - - - - - - - - - - > Returns first cookie with the given name or error
no
MultiPartReader()         - - - - - - - - - - > Returns a reader for parsing multipart form data
```

- - - - - *There are other things too but they are used internally*

Note: We'll learn more about each in the coming pages...

5.3. [Example]: Using `http.Request` to get request data

```
func yourHandler(w http.ResponseWriter, r *http.Request) {

    id := r.URL.Query().Get("id")
    ua := r.UserAgent()
    // etc
}
```

6. http.ResponseWriter Anatomy

6.1. Introduction

⇒ `http.ResponseWriter` is the interface you use in a handler to write an HTTP response.
⇒ Lets you write
    → Headers
    → Status Codes
    → Body
    → All in streaming manner

⇒ Passed to your handler by Go
⇒ `http.ResponseWriter` is an interface provided by Go's net/http package, not Struct

6.2. Core Structure of `http.ResponseWriter`

```
type ResponseWriter interface {
   Header() Header
   WriteHeader(statusCode int)
   Write([]byte) (int, error)
}

type Header map[string][]string

func (h Header) Add(key, value string) {...}
func (h Header) Set(key, value string) {...}

func (h Header) Get(key string) string {...}   - - used for http.Request (ignore here)
func (h Header) Values(key string) []string {...}  - - used for http.Request (ignore here)
```

1. Header() Header :

⇒ Returns Header map
    → Header is the NamedType with underlying type `map`
⇒ Lets you set response headers
⇒ **IMPORTANT:**
    → Must be called before the `Write()` and `WriteHeader()`
    → Because those methods lock the headers and send to to the client

2. WriteHeader(statusCode int) :

⇒ Marks headers as finalized and locked - - - - *Important to understand this*

⇒ Behind the scenes:

→ Locks headers and write it to the internal buffer

⇒ **IMPORTANT:**

→ If not called, default status code is 200

→ If you set header after this, that won't have any effect

3. Write([]byte) (int, error) :

⇒ Writes raw bytes to the response body

⇒ Triggers header send if not already sent

⇒ Behind the scenes:

→ Writes to an internal buffer

→ Flushes the buffer to TCP connection **when your handler returns**

⇒ **IMPORTANT:**

→ You can call it multiple times to write response body piece by piece

→ Once you start writing the body, headers are locked and sent.

→ ** *Learn more about Write in section:* '11.6.2. Writing Response with Raw Body' **

6.3. [Example]: Using `http.ResponseWriter` to write HTTP Response

```
func yourHandler(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/plain")
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("Hello, client!"))
}
```

NOTE	⇒ You can learn more about headers in its own section

7. HTTP Method & Protocol

⇒ As we see in the structure, the **`http.Request`** carries the Method and Protocols too

Note

In an HTTP request or response:
⇒ The Method, Path, and Protocol_Version are **part of the Start-Line.**
⇒ They are not part of the header section

Code Example

```
func yourHandler(w http.ResponseWriter, r *http.Request) {

        if r.Method == “POST”{
            // Handle for post method
        } else if r.Method == “GET” {
            // Handle for get method
        }

        if r.ProtoMajor == 2{
            // HTTP/2 - specific handling
        }

}
```

8. Query Parameters

www.server.com/path**?query1=value&query2=value**

8.1. Introduction

⇒ A URL (Uniform Resource Locator) defines the address of a resource on the web.
⇒ It consists of:
→ **Scheme:** http, https
→ **Host:** www.server.com
→ **Path:** /path
→ **Query Parameters** (optional): ?query1=value&query2=value

8.2. Introduction to `Request.URL` and `url.URL`

⇒ You don't need to manually split or parse URLs
→ Go does it for you
⇒ Every `**http.Request**` carries the `**URL**` in a fully parsed and populated url.
→ `**URL**` object inside `**http.Request**`

8.2. Core Structure of `url.URL` in `http.Request`

```
// http.Requestw
type Request struct {
    ...
    URL *url.URL
    ...
}

// url.URL
type URL struct {
   Path          string     ----------> "/users"
   RawQuery      string     ----------> "id=42&sort=asc"
}

func (u *URL) Query() Values {...}  ----------> Returns map with query params

type Values map[string][]string
```

* NOTE: there are many more things but these are the once you use *

8.3. [Example]: Using query parameters in handlers

Example URL: https://www.myserver.com/account/user**?name=gopher&age=22**

```
func handler(w http.ResponseWriter, r *http.Request) {
        path := r.URL.Path                  - - - - - - > "/account/user"
        rawQuery := r.URL.RawQuery    - - - - - - > "name=gopher&age=22"

        q := r.URL.Query()          - - - - - - - - - - - -> map of queries
        name := q.Get("name")      - - - - - - - - - - - -> "gopher"
        age := q.Get("age")          - - - - - - - - - - - -> "22"
}
```

9. WildCards (Path Parameters)

⇒ Wildcards let you capture dynamic parts of the URL path.
⇒ They are declared inside {} in route patterns
⇒ ServeMux does all of the things for you behind the scenes. You can just use it.

9.1. Wildcard forms

9.1.1. Single-segment wildcard {variable} ⇒ Exactly one path segment	9.1.2. Catch-all wildcard {all...} ⇒ Matches the remaining path
Example: "/user/**{name}**" **Example URL:** "/user/**goopher**" **Value of `name` = "goopher"**	**Example:** "/user/**{all...}**" **Example URL:** "/user/**goopher/other/path**" **Value of `all` = "goopher/other/path"**

9.2. Getting Wildcards Values in Handler

⇒ To handle the wildcards
⇒ We can use a `PathValue(string)` available in the `*http.Request`

```
// http.Request
type Request struct {
    ...
}

func (r *Request) PathValue(name string) string {...}
```

⇒ You can see the PathValue takes in the string(wildcard_name) and returns the (value)
⇒ Now you can use it like:

9.3. [Example]: Taking value from Path Param

```
mux.HandleFunc("/users/{id}", func(w http.ResponseWriter, r *http.Request) {
   id := r.PathValue("id")
   fmt.Fprintf(w, "User ID: %s", id)
})
```

Your Request URL: /users/**42**
Result: id = **42**

10. Headers

10.1. Introduction

⇒ Headers are metadata

→ key-value sent along with both HTTP requests and responses.

⇒ They describe things like:

→ How the data should be interpreted

→ Who is talking

→ Under what rules the communication happens

→ And other things too...

10.2. Types of Headers

⇒ Headers can be categorized by 2 types:

Direction	By role
→ Which side is the HTTP going (client to server or, server to client) → Request Headers and Response Headers	→ Role of the headers

10.3. Types of Headers by their Roles

10.3.1. Authentication and Authorization

⇒ Carries authentication stuffs

a. Authorization

→ Carries the authentication token to prove identity

→ Example: `**Authorization : Bearer your_auth_token_abc...**`

→ **Direction:** Client ➜ Server

b. WWW-Authenticate

→ Tells the client what kind of authentication the server expects after rejecting

→ Example: `**WWW-Authenticate : Bearer realm="api"**`

→ **Direction:** Server ➜ Client

10.3.2. Content & Data Representation

⇒ How the body is structured and interpreted

a. Content-Type

→Tells what Format is the Body

→ Example: **`Content-Type: application/json`**

→ **Direction:** Client ⇆ Server

(*We'll learn more detailed about it in its dedicated section*)

b. Content-Length

→ Tells Body size in bytes

→ Example: **`Content-Length: 348`**

→ **Direction:** Client ⇆ Server

c. Content-Encoding

→Tells how the body was compressed

→ Example: **`Content-Encoding: gzip`**

→ **Direction:** Client ⇆ Server

d. Content-Disposition

→Tells whether to display or download the content

→ Example: **`Content-Disposition: attachment; filename="a.pdf"`**

→ **Direction:** Server ➜ Client

e. Content-Language

→Tells the language of the body

→ Example: **`Content-Language: en-US`**

→ **Direction:** Server ➜ Client

10.3.3. Content Negotiation

⇒ Tells the server what kind of response do client prefer

a. Accept

→Tells server which response format the client can handle

→Example: **`Accept: application/json`**

→**Direction:** Client ➜ Server

b. Accept-Encoding

→Tells which compression methods the client supports

→Example: **`Accept-Encoding: gzip, br`**

→**Direction:** Client ➜ Server

c. Accept-Language

→Tells which language the client prefers

→Example: **`Accept-Language: en-US,en;q=0.9`**

→**Direction:** Client ➜ Server

10.3.4. State and Session (COOKIES)

(*We'll learn more detailed about it in its dedicated Cookies section* "**13. Cookies**")

⇒ Keeps user-specific state across multiple HTTP requests.

a. Cookie

→ Carries stored cookies from client to server

→ Example: **`Cookie: session_id=abc; theme=dark`**

→ **Direction:** Client ➜ Server

b.Set-Cookie

→ Instruction from server to client to store or update cookies

→ Example: **`Set-Cookie: session_id=abc; HttpOnly; Secure`**

→ **Direction:** Server ➜ Client

10.3.5. Caching & Validation

⇒ Controls when responses can be reused versus re-fetched

a. Cache-Control

→ Defines how long and what rules a response may be cached

→ Example: **`Cache-Control: no-cache, max-age=60`**

→ **Direction:** Client ⇆ Server

b. ETag

→ Identifies the exact version of a resource.

→ Example: **`ETag: "v2.3.1"`**

→ **Direction:** Server ➜ Client

c. If-None-Match

→ Asks the server if the resource version has changed or not

→ Example: **`If-None-Match: "v2.3.1"`**

→ **Direction:** Client ➜ Server

d. Last-Modified

→ Tells when the resource was last changed

→ **Direction:** Server ➜ Client

e. If-Modifies-Since

→ Ask the server if the resource changed after a given time.

→ **Direction:** Client ➜ Server

10.3.6. CORS (Browser Security)

⇒ Controls which origins are allowed to access server resources.

a. Origin

→ Tells the server where the request is coming from

→ Example: **`Origin: https://example.com`**

→ **Direction:** Client ➜ Server

b. Access-Control-Allow-Origin

→ Tells the browser which origins are permitted

→ Example: **`Access-Control-Allow-Origin: https://example.com`**

→ **Direction:** Server ➜ Client

c. Access-Control-Allow-Methods

→ Tells the browser which HTTP Methods are allowed cross-origin

→ Example: **`Access-Control-Allow-Methods: GET, POST`**

→ **Direction:** Server ➜ Client

d. Access-Control-Allow-Headers

→ Tells which request headers are allowed cross-origin

→ Example: **`Access-Control-Allow-Headers: Authorization, Content-Type`**

→ **Direction:** Server ➜ Client

Note

⇒ You can use these for debugging purpose too

10.3.7. Connection & Transport Control

⇒ Controls how long and how the underlying connection is used

a. Connection

→ Tells whether the TCP connection stays open or closes
→ Example: `**Connection: keep-alive**`
→ **Direction:** Server ➜ Client

b. Keep-Alive

→ Defines limits for reusing the open connection (which is kept alive)
→ Example: `**Keep-Alive: timeout=5**`
→ **Direction:** Server ➜ Client

c. Upgrade

→ **Requests or confirms** switching to another protocol
→ Example: `**Upgrade: websocket**`
→ **Direction:** Client ⇆ Server

10.3.8. Request Context & Metadata

⇒ Information about the request itself
⇒ Describes things like who is making the request and where it's going

a. Host

→ Tell the server which website/app the request is meant for.
→ or you can say it carries Domain Name
→ Example: `**Host: api.example.com**`
→ **Direction:** Server ➜ Client

b. User-Agent

→ Tells the server which client software is making the request
→ Example: `**User-Agent: Mozilla/5.0 (X11; Linux x86_64; r...**`
→ **Direction:** Server ➜ Client

c. Refer

→ Tells the server which page sent you here
→ Example: `**Referer: https://google.com/search?q=api**`
→ **Direction:** Server ➜ Client

10.4. Reading Headers from HTTP request

⇒ You can easily access Headers in your handlers with **`http.Request`**

10.5. Core Structure of Header

```
// http.Request
type Request struct {
    ...
    Header  Header
    ...
}

// URL structure
type Header map[string][]string

func (h Header) Get(key string) string {...}
func (h Header) Values(key string) []string {...}

func (h Header) Add(key, value string) {...}      - - - - used in testing / Client-Side
func (h Header) Set(key, value string) {...}      - - - - used in testing / Client-Side
```

* NOTE: there are many more things but these are the once you use *

You can see there are 2 ways to get the Headers

1. **Manual way**
   → Extract your header from the `Header` map
   → But you manually need to handle error and extract the the first element from the slice

2. **with Get()**
   → Does the slicing and error handling for you(so use this)

10.6. [Example]: Reading Headers from `http.Request`

```
func handler(w http.ResponseWriter, r *http.Request) {

    // Read single-value header
    contentType := r.Header.Get("Content-Type")
    auth := r.Header.Get("Authorization")

    // Since Headers are maps, you can do this too
    accepts := r.Header["Accept"]

}
```

10.7. Writing Headers to HTTP response

⇒ We've already covered this part in section "6. http.ResponseWriter Anatomy"

Enjoy Learning there in detailed ☺
Happy Learning

⇒ Now Let's move to the next topic

11. Body

11.1. Introduction

⇒ Body carries the actual payload/data in an HTTP request or response
⇒ Unlike headers, the body is actually data like JSON, form data, files, or plain text
⇒ Go exposes the body as stream to efficiently handle large payloads
⇒ We'll cover both **Request Body** and **Response Body** in detail

11.2. Body is a Stream (io.ReadCloser)

Request Body	Response Body
⇒ Data arrives in stream, not all at once ⇒ You read data sequentially ⇒ You close it when done	⇒ You write bytes to the client as stream ⇒ Behind the scene in go → data is buffered → and flushed to TCP connection

Learn deeper about it in these sections:
"2. Request Lifecycle inside Go net/http" *and* '2.4. Header and Body Streaming'

11.3. Types of Data in Body

⇒ There can be several types of a Body too.
⇒ The **type of body** data is defined by the **`Content-Type` header**

11.3.1. Raw Data :

⇒ Raw text data
⇒ Header: **`Content-Type: text/plain`**

11.3.2. JSON :

⇒ JSON structured data
⇒ Header: **`Content-Type: application/json`**

11.3.3. HTML Form :

⇒ Standard HTML form submissions (URL-encoded key-value pairs)
⇒ Header: **`Content-Type: application/x-www-form-urlencoded`**

11.3.4. Multipart Form :

⇒ More complex kind of body
⇒ Used for file uploads + form fields
⇒ Header: **`Content-Type: multipart/form-data`**

NOTE	⇒ Any type of body is raw bytes underneath. Header tells how to parse

11.4. Request Body

⇒ The **request body** is accessed through **`http.Request`**

Core Structure of http.Request

```
// http.Request
type Request struct {
    ...
    Body  io.ReadCloser      - - - - - - Just an interface
    ...
}

// io.Readcloser is the interface that groups the basic Read and Close methods.
type ReadCloser interface {
        Reader
        Closer
}

type Reader interface {
        Read(p []byte) (n int, err error)
}

type Closer interface {
        Close() error
}
```

Read() → **pulls the next chunk of bytes**
Close() → **releases underlying resources**

Critical Rule ! Don't Forget !

You must Close() the body after reading it

Why `io.ReadCloser` ?

⇒ Works uniformly for network connections, files, pipes, buffers.
⇒ Enables streaming with low memory footprints.

11.5. Response Body

Introduction

⇒ You can use `Write()` method present in `http.ResponseWriter`
→to send the response body to the client

Core Structure in http.ResponseWriter

```
type ResponseWriter interface {
   Header() Header
   Write([]byte) (int, error)        - - - - - - - - - Every type of response is sent through this
   WriteHeader(statusCode int)
}
```

What Write() does behind the scene

⇒ `Write()` does not mean "send bytes to the network right now."
⇒ Writes to an internal buffer
⇒ Flushes the buffer to TCP connection
⇒ Becomes HTTP response body
⇒ * *You can use* **Flush()** *to experiment and test this thing* *

IMPORTANT **Headers must be set before calling WriteHeader() or Write()**

! Don't Skip This !

⇒ You will be provided with different functions for sending different types of response body
⇒ But every function behind the scene is using this Write() method to send in bytes

⇒ They just parse from your datatype to raw bytes and send to client using Write()

So you've just learned to use a king of response

11.6. Raw Body (Foundation of every Body Type)

`Content-Type: text/plain`

⇒ Raw body = just raw bytes
⇒ Raw Body means, Everything is just uninterpreted bytes → []byte
⇒ Header: **`Content-Type: text/plain`**

! Don't Skip This !

⇒ Sending/Receiving raw body means working directly with bytes.
⇒ Which involves:
→ Converting your data types into bytes **before sending**,
→ Or parsing bytes back into your data types **after receiving.**

Now remember

→ Every other body type (JSON, forms, etc.) ultimately converts to raw bytes.

→ For example, a JSON object is stringified, converted to bytes, and sent as a raw body with the header Content-Type: application/json

So, once you grasping this you'll understand all body types

11.6.1. Handling Request with Raw Body [Foundation of Every Body]

(Foundation of every body reading)

What does "Reading Raw Body" means

⇒ Consume the entire request body stream
⇒ Convert it into a byte slice ([]byte)
⇒ Process bytes as needed.

How to do it

⇒ we use **`io.ReadAll()`** to get bytes from **`r.Body`**

Usecase Example

```
func MyHandler(w http.ResponseWriter, r *http.Request){

        bodyBytes , err := io.ReadAll(r.Body)   - - - - - - - Your any data will be in raw bytes
        defer r.Body.Close()

        // Things you do with body
}
```

Why use io.ReadAll() instead of r.Body.Read()?

⇒ You've seen `r.Body` has a `Read()` method, right?

⇒ So why not just use `r.Body.Read()` directly ?
→ Because Read() reads only the available chunks of stream, not full content
→ You'd need multiple calls and manual EOF checks
⇒ But:
→ `io.ReadAll()` calls `Read()` repeatedly **until EOF**
→ It collects and appends all chunks into a buffer.
→ Finally, it returns the complete byte slice.

Important NOTE: `io.ReadAll()` does not close the body; you must call `Close()` yourself

Why You must call r.Body.Close()

⇒ The request body is a stream, tied to a network connection
⇒ Calling `Close()` tells Go:
→ You're done reading
→ It can free buffers, file descriptors, and release the connection
→ Enables connection reuse (keep-alive)
⇒ Not closing leaks memory and connections

11.6.2. Writing Response with Raw Body

What does "Writing Raw Body" means

⇒ Convert your Data into raw bytes

⇒ And **send** those bytes to client **as a stream**

How to do it

⇒ use `Write()` from `http.ResponseWriter()`

⇒ Don't forget to use `Header` and `WriteHeader` too

Core Structure

```
type ResponseWriter interface {
   Header() Header
   Write([]byte) (int, error)
   WriteHeader(statusCode int)
}
```

What Write() does behind the scene

⇒ Writes to an internal buffer

⇒ Flushes buffer to TCP connection

⇒ Becomes HTTP response body

Code Example

```
func MyHandler(w http.ResponseWriter, r *http.Request){

        - - - - Always a good practice to write the headers manually
        w.Header().Set("Content-Type", "text/plain")
        w.WriteHeader(http.StatusOK) - - - -  - - - - - - -> optional here since 200 is
default

        var byteData []byte = []byte{"hello"}
        w.Write(byteData)
}
```

11.7. JSON BODY

`Content-Type: application/json`

11.7.1. Introduction

⇒ JSON is the most common format for structured data in HTTP request bodies.

11.7.2. How is JSON data handled in GO

Request Body	Response Body
⇒ Data arrives as stream ⇒ Data is bytes of JSON string ⇒ You parse bytes of JSON string to **Struct**	⇒ You decode your Struct to bytes of JSON string ⇒ Send it to client as stream

11.7.3. Different ways to handle JSON body

We have 2 ways to handle JSON body

1. Buffered

⇒ Request:

→First, read entire bytes of JSON string from the stream into a buffer

→then parse the buffer into a Go struct.

⇒ Response:

→ First, Convert the Go struct to bytes of JSON string first,

→ then write the entire buffer to the stream

2. Directly with Stream

⇒ Request:

→ Parse bytes of JSON string directly from the request stream into a Go struct as they arrive.

⇒ Response:

→ Encode the Go struct to bytes of JSON string while writing directly to the response stream as encoding happens.

** We'll learn both ways 🙂**

11.7.4. Go's `encoding/json` package(quick view)

⇒ Following the previous topic "Different ways to handle JSON body"
⇒ We have different json methods for them:

1. For Buffered way:

json.Unmarshal([]byte, any)	⇒ Parse bytes of JSON string into Go struct
json.Marshal(any) ([]byte, error)	⇒ Serialize Go struct to bytes of JSON string

2. For Directly with Stream way:

json.NewDecoder(io.Reader).Decode(any)	⇒ Decode JSON directly from stream into struct
json.NewEncoder(io.Writer).Encode(any)	⇒ Encode Struct to JSON writing directly to the stream

- - - - - - - - - ⇒ **Now let's see the coding part** ⇐ - - - - - - - - -

11.7.5. [Example]: Handling Request with JSON Body

Buffered Way

```
type Data struct {
   Name  string    `json:"name"`
   Age     int        `json:"age"`
}

func MyHandler(w http.ResponseWriter, r *http.Request){

        bodyBytes, _ := io.ReadAll(r.Body)
        defer r.Body.Close()            - - - - - - - - Don't forget to close the body

        var data Data
        err = json.Unmarshal(bodyBytes, &data)

        // Other things
}
```

Direct Parsing from Stream

```
type Data struct {
   Name  string    `json:"name"`
   Age     int        `json:"age"`
}

func MyHandler(w http.ResponseWriter, r *http.Request){

        defer r.Body.Close()            - - - - - - - - Don't forget to close the body

        var data Data
        json.NewDecoder(r.Body).Decode(&data)

        // Other things
}
```

Read Me	Understanding the Raw Body concept is essential for grasping how JSON data is handled, since JSON ultimately translates to raw bytes during reading and writing.

11.7.6. [Example]: Writing Response with JSON Body

Buffered Way

```
type Data struct {
   Name  string    `json:"name"`
   Age     int        `json:"age"`
}

func MyHandler(w http.ResponseWriter, r *http.Request){

        data := Data{Name: "Diwash", Age: 30}

        jsonBytes, err := json.Marshal(data)
        if err != nil {...}

        w.Header().Set("Content-Type", "application/json")  - don't forget to set headers
        w.Write(jsonBytes)
              |_____________ Since it's jsonBytes is raw bytes we are doing Write()

}
```

Direct Encoding to Stream

```
type Data struct {
   Name  string    `json:"name"`
   Age     int        `json:"age"`
}

func MyHandler(w http.ResponseWriter, r *http.Request){
        data := Data{Name: "Diwash", Age: 30}
        w.Header().Set("Content-Type", "application/json")  - - - don't forget to set headers
        json.NewEncoder(w).Encode(data)
}
```

Read Me	Understanding the Raw Body concept is essential for grasping how JSON data is handled, since JSON ultimately translates to raw bytes during reading and writing.

11.8. Multipart Formdata Body

`Content-Type: multipart/formdata`

11.8.1. Introduction

⇒ The `multipart/form-data` encoding type is used to send files + regular fields together to a server in a single HTTP request

Key Points:

⇒ The body is split into multiple parts
⇒ Each part has:
→ Its own headers
→ Its own content
⇒ Different parts can be text or binary(files)

- - - - - - - - ⇒ This is why it is more complex than JSON ** ⇐ - - - - - - - -

Note

This explains the raw structure well(*Strongly Recommended*):
https://medium.com/@muhebollah.diu/understanding-multipart-form-data-the-ultimate-guide-for-beginners-fd039c04553d

TIP

⇒ To see the original multipart form data you can do this:

```
func MyHandler(w http.ResponseWriter, r *http.Request) {
        data, _ := io.ReadAll(r.Body)
        fmt.Printf("The multipart form value is: %#v\n", string(data))
}
```

Read Me	Understanding the Raw Body concept is essential for grasping how JSON data is handled, since JSON ultimately translates to raw bytes during reading and writing.

11.8.2. Handling Request with Multipart Form Data

⇒ **`http.Request`** provides built-in helpers specifically for multipart data.
⇒ Some key methods and fields are:
→ **`r.MultipartForm`**
→ **`r.ParseMultipartForm()`**
→ **`r.FormValue()`**
→ **`r.FormFile()`**

NOTE: *We'll talk about each methods in coming pages*

11.8.2.1. Why not read multipart via `r.Body`

In-short:
⇒ Just like Go provides NewEncoder() and NewDecoder() for JSON,
⇒ it also offers dedicated methods for handling Multipart FormData.

In-long:
⇒ r.Body is a raw stream of bytes.
⇒ For eg: For JSON, you read + unmarshal

⇒ Multipart form data is more complex.
⇒ So for that is providing you those helpers for:
→ automatic parsing
→ memory + disk handling
→ clean APIs
⇒ So use them

11.8.2.2. Core Structure in http.Request

```
// http.Request
type Request struct {
    ...
    MultipartForm *multipart.File
    ...
}

func (r *Request) ParseMultipartForm(maxMemory int64) error {...}
func (r *Request) FormValue(key string) string {...}
func (r *Request) FormFile(key string) (multipart.Form, *multipart.FileHeader, error)

// multipart.Form
type Form struct {
        Value map[string][]string                  ------- Normal Fields
        File  map[string][]*FileHeader             ------- Files
}

// multipart.File
type File interface {          --------- Just a normal stream with a little more powers
        io.Reader
        io.ReaderAt
        io.Seeker
        io.Closer
}

// multipart.FileHeader
type FileHeader struct {
        Filename string
        Header   textproto.MIMEHeader
        Size     int64

        content   []byte        ---- Main binary file(small file in memory, large file in disk)
        tmpfile   string
        tmpoff    int64
        tmpshared bool
}

            * NOTE: there are many more things but these are the once you use *
```

ParseMultipartForm(maxMemory int64) error :

⇒ Reads the entire multipart form body from r.Body
⇒ Parses into form fields and files
⇒ Stores:
→ Small Fields and Files in memory (up to maxMemory bytes)
→ Large Files in temporary disk
→ * Size depends upon `maxMemory` that you gave
⇒ Populates `r.MultipartForm` with parsed data:
→ `r.MultipartForm.Value` for fields
→ `r.MultipartForm.File` for files

MultipartForm *multipart.FormError :

⇒ The parsed, structured representation of multipart data
⇒ Accessible only after ParseMultipartForm is called
⇒ Fields:
→ `r.MultipartForm.Value` // map[string][]string
→ `r.MultipartForm.File` // map[string][]*multipart.FileHeader

⇒ **IMPORTANT NOTES:**
→ Only available after you run `ParseMultipartForm(maxMemory)`
→ Typically used for full manual control, rarely needed directly.

FormValue(key string) string :

⇒ Returns the first value for the given key from form fields.

⇒ **IMPORTANT NOTES:**
→ Works for both **`application/x-www-form-urlencoded`** and **`multipart/form-data`** and even URL queries

FormFile(key string) (multipart.File, *multipart.FileHeader, error) :

⇒ Retrieves the file associated with the given key.
⇒ Returns:
→ `multipart.File`: file stream interface
→ `*multipart.FileHeader`: metadata about the file (filename, size, headers)
→ `err` if file is not found or inaccessible

⇒ **IMPORTANT NOTES:** * *We'll learn about File things in its dedicated section**

11.8.2.4. Accessing Value from Multipart Form Data request

⇒ We saw there is `MultipartForm` which has field `Values` for values

⇒ But we saw one method too, `FormValue(key)`, which provides you the value too

→ We use `FormValue()` because it does the slicing and error handling thing for you.

⇒ Let's see both's example:

Example

```
func MyHandler(w http.ResponseWriter, r *http.Request) {
        r.ParseMultipartForm(32 << 10)

        name1 := r.MultipartForm.Value["name"][0] - - - - - - because it contains slice
        name2 := r.FormValue("name")

}
    * NOTE: there are many more things if you tweak but these are the once you use *
```

11.8.2.5. Accessing File from Multipart Form Data request

⇒ Use `r.FormFile(key)` to get the uploaded file associated with the form field name key

Example

```
func MyHandler(w http.ResponseWriter, r *http.Request) {
        err := r.ParseMultipartForm(32 << 10)
        if err != nil {...}

        name1 := r.MultipartForm.Value["name"][0] - - - - - - because it contains slice
        file, fileHeader, err := r.FormFile("uploadfile")

        fmt.Printf("Uploaded File: %+v\n", fileHeader.Filename)
        fmt.Printf("File Size: %d bytes\n", fileHeader.Size)
        fmt.Printf("MIME Header: %+v\n", fileHeader.Header)
        - - - and many more...

}
    * NOTE: there are many more things if you tweak but these are the once you use *
```

Note	* We'll learn about handling Files in its own dedicated section *

11.9. HTML Form Body

`Content-Type: application/x-www-form-urlencoded`

11.9.1. Introduction

⇒ It is the default encoding for HTML forms. Sends data as URL-encoded key-value pairs in the request body.

Example

⇒ The bytes version of this string:

```
name=goopher&age=30&planet=Earth
```

How it works

⇒ Keys and values are URL-encoded

⇒ Example:

→name = John Doe → becomes → “name=John+Doe”

→ message = hello&bye → becomes→ “hello%26bye”

⇒ You can learn more about “x-www-form-urlencoded” encoding type here:
https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods/POST

11.9.2. Handling Request with HTML Form body

⇒ `http.Request` provides built-in helpers specifically for html form.
⇒ Some key methods and fields are:
→ `r.ParseForm()`
→ `r.FormValue()`
→ `r.PostFormValue()`

11.9.3. Why not just r.Body()?

In-short:
⇒ Same reason you don't manually parse JSON bytes when json.Decoder exists.

In-long:
⇒ r.Body is a raw stream of bytes.
⇒ For eg: For JSON, you read + unmarshal
⇒ HTML form has its own encoding type too which is mess to decode manually
⇒ So for that is providing you those helpers for:
→ Read the raw body
→ parse from encoding
→ Store results in structured maps
⇒ So use them

11.9.4. Core Structure in http.Request

```
// http.Request
type Request struct {
    ...
    Form url.Values
    PostForm url.Values
}

func (r *Request) ParseForm() error {...}
func (r *Request) FormValue(key string) string {...}
func (r *Request) PostFormValue(key string) string {...}

// url.Values
type Values map[string][]string
```

** NOTE: there are many more things but these are the once you use **

- - - - - - - - ⇒ **Now Let's See what each methods do** ⇐ - - - - - - - -

ParseForm() error :

⇒ Reads the entire request body from r.Body(stream)
⇒ Parses the HTML encoded values
⇒ **Populates** following fields with parsed data:
→ `r.Form` - - - - - query params + POST body
→ `r.PostForm` - - - - - same as Form but for POST request only

⇒ **IMPORTANT NOTE:**
→ You cannot access `r.Form` and `r.PostForm` without calling `ParseForm()`

FormValue(key string) string :

⇒ Returns the first value for the given key from `r.Form`

⇒ **IMPORTANT NOTE:**
→ Works for both **`application/x-www-form-urlencoded`** and **`multipart/form-data`** and even URL queries

PostFormValue(key string) string :

⇒ Same as FormValue but for POST request body only

11.9.6. Accessing Form Values

⇒ There can be 2 types you extract the value

1. Manually:
→ Directly access from `r.Form` → `map[string][]string`

2. FormValue(Key):
→ Use `FormValue(key)`
→ Does the slicing and error handling things for you.

⇒ Let's see both's example:

Example

```
func MyHandler(w http.ResponseWriter, r *http.Request) {
        r.ParseForm()                   ------ don't forget to parse the form

        name1 := r.Form["name"][0]      ------ because it contains slice
        name2 := r.FormValue("name")    ------ recommended
}
    * NOTE: there are many more things if you tweak but these are the once you use *
```

12. Request Context

12.1. Before we start

⇒ This assumes you already understand Go's context package in a really good way.
⇒ We're not relearning context.
⇒ We're learning how it's used in backend HTTP flow

12.2. Introduction

⇒ Context is a way to carry **request-scoped data** and **cancellation signals** throughout an **HTTP request's lifecycle**
⇒ Context in Backend is like control tower for:
→ cancellation signals
→ timeouts/deadlines
→ request-scoped values

12.3. Where does request context come from?

⇒ Go's net/http server automatically creates a context per incoming request
⇒ This context is owned by the HTTP server, **not you**
⇒ It is tightly bound to the client connection lifecycle

Which means:

- Client disconnects → context is canceled
- Server times out → context is canceled
- Handler returns → request is done

12.4. Context Structure in http.Request

```
type Request interface {
    ...
}

func (r *Request) Context() context.Context
func (r *Request) WithContext(ctx context.Context) *Request
```

→ That's it.
→ Everything else is built on top of these two.

Context() context.Context :

⇒ Returns the current context attached to the `http.Request`

Key Properties:

⇒ Immutable
⇒ Controlled by `net/http`
⇒ Linked to client connection
⇒ Automatically canceled on disconnect or timeout

Important:

⇒ You should never always work on the root context provided by default
⇒ You should always derive from it

WithContext(ctx) *Request :

⇒ Returns a shallow copy of the request with a different context attached provided by you

Key Points:

⇒ Doesn't do the hard copy
⇒ Cancellation and deadline still propagate
⇒ Only swaps the Context field with your given Context

Main use:

⇒ Create your new context with root context given by http.Request.Context()
⇒ You mainly do this in middlewares
⇒ Pass it down to the handlers

BIG NOTE

Always remember to listen to the http.Request.Context() cancel signals

12.6. [Example]: Handling Context in Request

```
func main() {
        var mux *http.ServeMux = http.NewServeMux()
        mux.Handle("/", Middlewares(MyHandler))
        http.ListenAndServe(":8080", mux)
}

type ContextKey string
var UserId ContextKey = "UserId"

func Middlewares(nextFunc func(http.ResponseWriter, *http.Request)) http.Handler {

    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        oldCtx := r.Context()          - - - - Taking old context
        ctx := context.WithValue(oldCtx, UserId, "ID_CODE_123_abc")
        req := r.WithContext(ctx)    - - - - Shallow copy of old request with new ctx

        nextFunc(w, req)
    })
}

func MyHandler(w http.ResponseWriter,  r *http.Request) {
    var userId any = r.Context().Value(UserId)   - - Will get context passed by middleware
    fmt.Printf("The user id is: %#v", userId)
}
```

Here's what we're doing

- - - - - *In Middleware*

1. We extract the original request context using `r.Context()`
→ This is the context created by Go's HTTP server
→ It is tied to the client connection

2. We derive a new context using `context.WithValue()`
→ Parent context stays intact
→ Cancellation and deadline still propagate
→ We only added request-scoped data

3. We create a shallow copy of `http.Request`

4. We pass the new request forward

- - - - - *In Handler*

5. We read the value from `r.Context()`
→ Using the same typed key (UserId)

12.7. [Important] : Why use Named Type variables for Context keys

Recommended

```
type ContextKey string
const UserIDKey ContextKey = "userID"

func Middleware(next http.Handler) http.Handler {
        ...
        ctx := context.WithValue(r.Context(), UserIDKey, "12345")
        ...
}
```

Not Recommended

```
const UserIDKey = "userID"

func Middleware(next http.Handler) http.Handler {
        ...
        ctx := context.WithValue(r.Context(), UserIDKey, "12345")
        ...
}
```

👇 - - - - - - - - ⇒ **Let's Understand Why In Next Page** ⇐ - - - - - - - - 👇

12.7.1. Starting from foundation

⇒ Go is a statically typed language
⇒ So when you use `==` operator
→ Go compares both **Type** and **Value**

→ Which means:
→ It doesn't check just the value
→ It checks the type too
→ If the type doesn't match, it falls. even if the value matches

Example

```
type CtxKey string
var NamedCtxKey CtxKey = "userId"
var NormalCtxKey string = "userId"

func(type1 any, type2 any) {
        if type1 == type2 {                    - - - - -  Doesn't show the error but returns false
                fmt.Println("It matched...")
        }
        fmt.Println("It didn't matched...")
}(NamedCtxKey, NormalCtxKey)

if NamedCtxKey == NormalCtxKey {    - - - - Shows error cause type don't match
        print("matched here too")
}
```

12.7.2. How Context compares keys internally

⇒ When calling `ctx.Value(key)`, Go does:
→ `storedKey == providedKey`

⇒ Since `==` compares **type + value**, keys must match exactly in both
→ If both **type** and **value** doesn't match, lookup returns nil

12.7.3. So why to use the Named type?

⇒ Different packages may use the same key string like "**userId**"
⇒ Plain strings cause silent collisions and overwrites
⇒ Named types ensure unique key types, preventing collisions even if values match

13. Cookies

This section assumes that you already know what cookies

13.1. Introduction

⇒ Cookies are HTTP headers but complex to parse manually.
⇒ For eg:

HTTP Request with Cookies :

```
GET /dashboard HTTP/1.1
Host: example.com
Cookie: session_id=abc123; theme=dark; lang=en-US

...
```

HTTP Response with Cookies :

```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: session_id=abc123; Path=/; HttpOnly; Secure; Max-Age=3600
Set-Cookie: theme=dark; Path=/; Max-Age=86400
Set-Cookie: lang=en-US; Path=/; Max-Age=86400
...
```

BUT

⇒ Even though cookies are headers, parsing them manually is complicated.
→ Because you can see its structure is not normal
⇒ So, you don't handle them as plain headers in Go backend.

⇒ In http request:
→ `net/http` parses cookies into structured object `Cookie`
⇒ For http request:
→ `net/http` provides you methods to write from structured object

⇒ So, Let's see it's core structure

13.2. Core Structure of Cookie

```
        * - - - - - - - - - - - - - - Main Skeleton of Cookie in Go - - - - - - - - - - - - - - *
// http.Cookie
type Cookie struct {
        Name   string
        Value  string
        Quoted bool // indicates whether the Value was originally quoted

        Path       string    // optional
        Domain     string    // optional
        Expires    time.Time // optional
        RawExpires string    // for reading cookies only

        // MaxAge=0 means no 'Max-Age' attribute specified.
        // MaxAge<0 means delete cookie now, equivalently 'Max-Age: 0'
        // MaxAge>0 means Max-Age attribute present and given in seconds
        MaxAge      int
        Secure      bool
        HttpOnly    bool
        SameSite    SameSite
        Partitioned bool
        Raw         string
        Unparsed    []string // Raw text of unparsed attribute-value pairs
}

                    * - - - - - - - - - - - - - - Request Part - - - - - - - - - - - - - - *
// http.Request
type Request struct {
   ...
}

func (r *Request) Cookie(name string) (*Cookie, error) {...}
func (r *Request) Cookies() []*Cookie {...}


                    * - - - - - - - - - - - - - - Response Part - - - - - - - - - - - - - - *
// http.SetCookie
func SetCookie(w ResponseWriter, cookie *Cookie) {...}
```

1. Cookie :

⇒ Main Structured representation of a cookie

⇒ In request:

→ Go automatically populates it for you.

⇒ In response:

→ you create a `Cookie` struct and write it with `SetCookie(w, *Cookie)`

2. r.Cookie(cookie_name) :

⇒ Use this to extract a cookie by name from the request

3. r.Cookies() []*Cookie :

⇒ Extracts all cookies from the request

4. SetCookie(w, *Cookie) :

⇒ Takes http.ResponseWriter and *Cookie, writes the cookie into the response

13.3. [Example]: Reading Cookies From Request

```
func handler(w http.ResponseWriter, r *http.Request) {
    var err error

    var c *http.Cookie
    c, err = r.Cookie("session_id")
    if err != nil {
        ...
    }

    var cookies []*http.Cookie
    cookies = r.Cookies()
    for _, cookie := range cookies {
        fmt.Printf("Cookie %s = %s\n", cookie.Name, cookie.Value)
    }
}
```

REMEMBER: it's `http.Cookie` not `r.Cookie`

13.4. [Example]: Giving Cookies As Response

```
func handler(w http.ResponseWriter, r *http.Request) {
    var cookie1 http.Cookie
    var cookie2 http.Cookie

    cookie1 = http.Cookie{
        Name:     "session_id",
        Value:    "abc123",
        Path:     "/",
        HttpOnly: true,
        Secure:   true,
        MaxAge:   3600,
        SameSite: http.SameSiteLaxMode,
    }
    http.SetCookie(w, &cookie1)

    cookie2 = http.Cookie{
        Name:   "theme",
        Value:  "dark",
        Path:   "/",
        MaxAge: 86400,
    }
    http.SetCookie(w, &cookie2)
}
```

REMEMBER: it's `http.Cookie` not `r.Cookie`

14. Middlewares

14.1. Introduction

⇒ Middleware is a logic which runs before your HTTP handlers

⇒ Middleware are just Functions
⇒ Which takes **Handler** as a Parameter
⇒ And returns a **new Handler** which:
→ Runs the middleware logic
→ and calls that **Handler**

14.2. [Example] : Basic Middleware

```
func main() {
        var mux *http.ServeMux = http.NewServeMux()
        mux.Handle("/", AuthMiddleware(YourHandler))
        http.ListenAndServe(":8080", mux)
}

type ContextKey string
var UserId ContextKey = "UserId"

func AuthMiddleware(nextFunc func(http.ResponseWriter, *http.Request))
http.Handler {

        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {

                *- - - - - - - Middleware logic - - - - - - - *
                ctx := context.WithValue(r.Context(), UserId, "ID_CODE_123_abc")
                req := r.WithContext(ctx)

                *- - - - - - - Calling the next Function - - - - - - - *
                nextFunc(w, req)

        })
}
```

** Look Care Fully **

14.3. Multiple Middleware - Handle Scoped

⇒ While Applying Middlewares to a single Handler

→ You apply multiple middleware to a single route

→ Each middleware wraps the next handler

```
func main() {
        var mux *http.ServeMux = http.NewServeMux()
        mux.Handle("/", Middleware_1(Middleware_2(YourHandler)))
        http.ListenAndServe(":8080", mux)
}
```

NOTE

⇒ You are applying multiple middlewares to a single handler

⇒ This affects only that route

14.4. Multiple Middleware - Global

We already know:

⇒ Middleware are wrappers of Handlers

⇒ and mux is also a handler acting like a middleware

Therefore

⇒We can wrap mux with middleware too

```
func main() {
        var mux *http.ServeMux = http.NewServeMux()

        mux.HandleFunc("/hello", HelloHandler)
        mux.HandleFunc("/admin", AdminHandler)

        wrappedMux := Middleware_1( Middleware_2(mux))

        http.ListenAndServe(":8080", wrappedMux)
}
```

This means:

⇒ Middleware runs before routing

⇒ Applies to every route automatically

Confused about mux?

→ check section '4.6. [IMPORTANT] : Deep Dive into http.ServeMux and http.Server'

15. Database(pgx)

15.1. Introduction

⇒ Go values simplicity, clarity, and control
⇒ Go does not push ORMs by default
⇒ Go encourages writing raw DB Queries
⇒ This gives:
→ Full control over database logic
→ Predictable performance
→ No hidden behavior

⇒ In this diary, we learn PostgreSQL with Go, the Go way.

15.2. What "Connecting to a Database" Actually Means

⇒ Go itself cannot talk to databases
⇒ "Connecting to PostgreSQL" means:
→ Creating a network session between your Go program and PostgreSQL

⇒ At lowest level:
→ PostgreSQL is a separate server
→ Communication happens over TCP
→ SQL is sent as bytes
→ Results come back as structured responses
⇒ So to talk to PostgreSQL, Go needs something that:
→ Opens TCP connections
→ Speaks PostgreSQL wire protocol
→ Encodes Go values into bytes
→ Decodes bytes back into Go values
→ Exposes a usable Go API

15.3. The Role of a Database Driver

⇒ A database driver sits between:
- → Your Go code
- → The PostgreSQL server

⇒ The driver is responsible for:
- → Network communication
- → Authentication
- → Query execution
- → Encoding and decoding data
- → Managing connection state

⇒ Go does not do this by itself

⇒ This is why drivers exist

15.4. Introducing pgx

⇒ pgx is a **PostgreSQL-native client and driver** written for Go

⇒ pgx is responsible for everything between your code and PostgreSQL

⇒ pgx handles:

- → PostgresSQL wire protocol
- → TCP connection lifecycle
- → Query execution
- → Prepare statements
- →Transactions
- → Type encoding and decoding
- → Connection pooling (via pgxpool)
- → PostgreSQL-specific features

15.5. Connection Management Models in pgx

⇒ There is only one way to connect to PostgreSQL.

⇒ But there are two ways to manage connections:

1. **Single Connection (pgx.Conn)**
2. **Concurrent connections using pool (pgxpool.Pool)**

Everything builds on top of one fundamental unit: **Single Connection (pgx.Conn)**

** Let's learn about them **

15.6. Single Connection Model (pgx.Conn)

Read Me

You won't normally use it but you should know this since everything is built on top of this

15.6.1. What a Single Connection Is

⇒ A single connection means:

→ One TCP socket
→ One Postgresql session
→ One execution timeline

Now Read Carefully ↴

⇒ So it has Limitations like:

→ Can execute only one operation at a time
→ Is NOT concurrent-safe
→ Must be manually opened and closed

15.6.2. Creating a Single Connection

```
conn, err := pgx.Connect(ctx, os.Getenv("DATABASE_URL"))
if err != nil {...}
defer conn.Close(ctx)
```

⇒ What `pgx.Connect` does:

→ Opens a TCP connection
→ Authenticates with PostgreSQL2233
→ Starts a session
→ Returns `pgx.Conn` *< - - - - - - we'll see its core structure in next page*

⇒ You'll run all queries using this connection (`conn`)

15.6.3. Core structure of `pgx.Conn`

```
type Conn struct {
        ...
}

func (p *Conn) Query(ctx context.Context, sql string, args ...any) (Rows, error)
func (p *Conn) QueryRow(ctx context.Context, sql string, args ...any) Row
func (p *Conn) Exec(ctx context.Context, sql string, args ...any) (pgconn.CommandTag,
error)
func (p *Conn) Begin(ctx context.Context) (Tx, error)
func (p *Conn) Acquire(ctx context.Context) (*pgxpool.Conn, error)
func (p *Conn) Close()

type Row interface {
        Scan(dest ...any) error
}

type Rows interface {
        Scan(dest ...any) error
        Next() bool
        Close()
        Err() error
        Conn() *Conn
        ...
}

type Tx interface {
        Query(ctx context.Context, sql string, args ...any) (Rows, error)
        QueryRow(ctx context.Context, sql string, args ...any) Row
        Commit(ctx context.Context) error
        Exec(ctx context.Context, sql string, arguments ...any) (commandTag
        pgconn.CommandTag, err error)
        Rollback(ctx context.Context) error
}
```

Close()

⇒ Closes the TCP socket

⇒ Ends PostgreSQL session

⇒ Frees server resources

⇒ Conn becomes unusable forever

Query(ctx, sql, args...)

⇒ Execute a query returning multiple rows

⇒ **Rows are streamed** from PostgreSQL

→ Streamed row-by-row

→ Not all at once

⇒ Connection is locked until rows are closed (**Close()**)

⇒ You must:

→ Fully iterate

→ or, Call `**rows.Close()**`

⇒ IMPORTANT:

→ While rows are open, no other query can run on this connection

→ So you must `**rows.Close()**`

QueryRow(ctx, sql, args...)

⇒ Executes query returning one row

⇒ Don't have iterator or cursor like in `**Query()**`

⇒ **Query is actually sent when you do Scan(), not before**

⇒ Locks the connection until you call `**Scan()**`

⇒ It behaves this way:

→ 0 rows → pgx.ErrNoRows

→ >1 rows → first row taken, rest discarded

Exec(ctx, sql, args...) :

⇒ Executes statements with no returned rows

⇒ Returns CommandTag

⇒ Used for INSERT / UPDATE / DELETE without RETURNING

Begin(ctx) :

⇒ Start a transaction on this connection

⇒ Transaction locks the connection

⇒ No other queries allowed until:

→ Commit

→ or Rollback

⇒ WARNING: Forget roll back poisons the connection

⇒ Transactions are always connection-scoped

Ping(ctx) :

⇒ Checks if connection is alive

⇒ Verifies socket + server responsiveness

⇒ Mostly used by pools and health check

15.7. Concurrent Connection Model (pgxpool.Pool)

Read Me

Built on top of single connection (pgx.Conn)

15.7.1. Why pgxPool Exists

Problems with pgx.Conn:

⇒A single `pgx.Conn` can handle only one operation at a time

⇒ While a query, rows, or transaction is active, the connection is blocked

⇒ Not safe for concurrent Request

Solution you might think of:

⇒ To handle multiple request, Create new connections per request

⇒ But it's bad because:

→ Opening TCP connections is expensive

→ PostgreSQL enforces connection limits (so you'll have to manage that)

→ Too many connections crash performance

Real solution: pgxpool

⇒ pgxpool is a manager of multiple `pgx.Conn`s

⇒ It manages things like:

→ Connection limits

→ **Reusing** existing connections

→ Safely sharing connections across requests

Very Important Note

⇒ pgxpool is not a separate system from pgx.Conn
⇒ pgxpool does not replace pgx.Conn

⇒ It is built directly on top of pgx.Conn
⇒ It manages multiple pgx.Conn
⇒ All queries still run on individual pgx.Conns

Makes sense right ?

15.7.2. [Example]: Creating a Pool

```
pool, err := pgxpool.New(ctx, os.Getenv("DATABASE_URL"))
if err != nil {...}
defer pool.Close()
```

pgxpool.New() :

→ Creates a new pool object
→ Doesn't create a "main connection"
→ Returns **pgxpool.Pool**
→ Lazily opens and manages multiple ***pgx.Conn**

15.7.3. Core structure of pgxpool.Pool

```
type Pool struct {
        ...
}

func (p *Pool) Query(ctx context.Context, sql string, args ...any) (Rows, error)
func (p *Pool) QueryRow(ctx context.Context, sql string, args ...any) Row
func (p *Pool) Exec(ctx context.Context, sql string, args ...any) (pgconn.CommandTag,
error)
func (p *Pool) Begin(ctx context.Context) (Tx, error)
func (p *Pool) Acquire(ctx context.Context) (*pgxpool.Conn, error)
func (p *Pool) Close()

type Row interface {
        Scan(dest ...any) error
}

type Rows interface {
        Scan(dest ...any) error
        Next() bool
        Close()
        Err() error
        Conn() *Conn
        ...
}

type Tx interface {
        Query(ctx context.Context, sql string, args ...any) (Rows, error)
        QueryRow(ctx context.Context, sql string, args ...any) Row
        Commit(ctx context.Context) error
        Exec(ctx context.Context, sql string, arguments ...any) (commandTag
        pgconn.CommandTag, err error)
        Rollback(ctx context.Context) error
}
```

⇒ **You can see:**

→ It as save methods like `pgx.Conn`

→ But the execution is different, since it has to manage multiple connections too

****** Let's now understand how those methods work different than `pgx.Conn` ******

15.7.4. How Queries Operate Inside a Pool

When you run a query using pool:
Step 1: Asks the pool for a connection
Step 2: Pool checks if it has a free connection or not

CASE 1: A free connection
⇒ Pool takes one unused **`*pgx.Conn`**
⇒ Marks it as “inuse”
⇒ Hands it to your query
⇒ That connection runs **`QueryRow`** of **`*pgx.Conn`**
⇒ **`Scan()`** reads the result
⇒ Query finishes
⇒ Pool marks the connection as “free again”
⇒ Connection stays open and goes back to pool

CASE 2: No free connection, BUT pool has not reached max limit
⇒ Pool sees all existing connection are busy
⇒ Pool sees connection !< max
⇒ Creates a new connection **`*pgx.Conn`**
⇒ Query runs on it
⇒ Result scanned
⇒ Connection stays open and goes to pool for future use

CASE 3: No free connection AND poll is at max limit
⇒ Pool sees all existing connections are busy and max connection limit as reached
⇒ Your goroutine waits
⇒ Nothing crashes
⇒ As soon as any other query finishes:
⇒ That connection is returned to the pool
⇒ Your goroutine gets the connection
⇒ Your query runs
⇒ If ctx times out → error returned.

What if waiting takes too long?
⇒ That’s where **`context.Context`** comes in
⇒ If context times out or the request is canceled
→ Pools stops waiting
→ Returns an error
→ No connection is used

15.8. Queries (Practical Examples)

⇒ In this section we'll now finally learn "real ways to query things from databases".
⇒ We've learned in previous sections that `pgx.Conn` and `pgxpool.Pool` has same methods for same things but the mechanism is different.

→ So here we'll be learning only from the `pgxpool.Pool` side but you can apply the same thing in `pgx.Conn` too if you are using that

15.8.1. [Example]: Query a Single Row

⇒ We use `QueryRow` method provided by `pgxpool.Pool` / `pgx.Conn`

Learn more about it in section "15.6.3. Core structure of `pgx.Conn`"

```
type Employee struct {
	ID	int
	FirstName string
	LastName  string
}

func YourFunction(pool *pgxpool.Pool) {

	var employee Employee

	row := pool.QueryRow(ctx,
		"SELECT id, first_name, last_name FROM employee LIMIT 1;",
	)

	err = row.Scan(&employee.ID, &employee.FirstName, &employee.LastName)
	if err != nil {
		panic(err)
	}

	fmt.Printf("The row is: %#v", employee)
}
```

15.8.2. [Example]: Query Multiple Rows

⇒ We use `Query` method provided by `pgxpool.Pool` / `pgx.Conn`

Learn more about it in "15.6.3. Core structure of `pgx.Conn`" section

```
func QueryMultipleRows(ctx context.Context, pool *pgxpool.Pool) {
    rows, err := pool.Query(ctx, "SELECT id, first_name, last_name FROM employee;")
    if err != nil {
        panic(err)
    }
    defer rows.Close()

    var employees []Employee
    for rows.Next() {
        var emp Employee
        err = rows.Scan(&emp.ID, &emp.FirstName, &emp.LastName)
        if err != nil {
            panic(err)
        }
        employees = append(employees, emp)
    }

    if rows.Err() != nil {
        panic(rows.Err())
    }

    fmt.Printf("Employees: %#v\n", employees)
}
```

15.8.3. [Example]: Executing Non Returning Commands

⇒ We use `Exec` method provided by `pgxpool.Pool` / `pgx.Conn`

Learn more about it in section '15.6.3. Core structure of `pgx.Conn`'

```
func ExecCommand(ctx context.Context, pool *pgxpool.Pool) {
        commandTag, err := pool.Exec(ctx, "UPDATE employee SET last_name=$1 WHERE
        id=$2", "Smith", 1)

        if err != nil {
                panic(err)
        }
        fmt.Printf("Rows affected: %d\n", commandTag.RowsAffected())
}
```

15.9. Transactions

⇒ For transactions we use the **"Begin()"** method provided by pgx.
⇒ **`Begin(ctx)`** starts a transaction and returns a **`Tx`** object

15.9.1. Core Structure of `Tx`

```
type Tx interface {
        Query(ctx context.Context, sql string, args ...any) (Rows, error)
        QueryRow(ctx context.Context, sql string, args ...any) Row
        Commit(ctx context.Context) error
        Exec(ctx context.Context, sql string, arguments ...any) (commandTag
pgconn.CommandTag, err error)
        Rollback(ctx context.Context) error
}
```

⇒ Connection is blocked until you run **`tx.Commit()`** or **`tx.Rollback()`**

NOTE: **`Rollback()`** cancels the whole transaction and returns the database to the state before BEGIN. It ignores savepoints.

15.9.2. [Example]: Handling Transactions with pool

```
func TransactionExample(ctx context.Context, pool *pgxpool.Pool) {

    tx, err := pool.Begin(ctx)
    if err != nil {
        panic(err)
    }

    defer func() {
        if err != nil {
            tx.Rollback(ctx)
            return
        }
        err = tx.Commit(ctx)
    }()

    var firstName string
    err = tx.QueryRow(ctx, "SELECT first_name FROM employee WHERE id=$1",
1).Scan(&firstName)
    if err != nil {
        return
    }
tm
    fmt.Printf("Updated employee first name: %s\n", firstName)
}
```

NOTE: `Rollback()` cancels the whole transaction and returns the database to the state before BEGIN. It ignores savepoints.

15.10. Avoiding SQL Injections

⇒ You can just see and understand this way is really bad to write query with dynamic values:

```
query := "SELECT * FROM users WHERE email = '" + email + "'"
row := pool.QueryRow(ctx, query)
```

⇒ Easily SQL injectable

Our solution ⇒ Parameterized Query

```
pool.Exec(ctx,
    "INSERT INTO users (email, age) VALUES ($1, $2)",
    email,
    age,
)
```

How it works behind the scene:
⇒ Sends SQL string to the PostgreSQL
⇒ Sends parameter values separately
⇒ PostgreSQL:
→ Parses SQL syntax
→ Builds execution plan
→Then only, injects values into placeholders safely

Big NOTE

⇒ SQL parameters works only for values
⇒ They don't work for:
→ Table names
→ column names

16. Authentication

⇒ In the authentication part we will be learning 2 methods of authentications:
→ “JWT”
→ “OAuth”

⇒ This chapter assumes you already know following things:
→ What authentication is
→ What are JWT and OAuth

16.1. JWT Authentication

⇒ To handle JWT we use a package called **“golang-jwt/jwt”**
⇒ It manages things like:
→ creating JWT tokens
→ parsing JWT strings
→ validating signature and standard claims(not custom claims)
→ Providing flexible claims types (map or structs)

⇒ Official docs (recommended reading):

https://golang-jwt.github.io/jwt/
https://golang-jwt.github.io/jwt/usage/create/

NOTE

⇒ This section is kind of confusing so better practice codes and check the source code while you’re learning from this book

16.1.1. Core Structure of `golang-jwt`

```
                    * - - - - - - Parsed Structured form of JWT string - - - - - - *
type Token struct {
  Raw       string                        - - - - - - - - -  raw JWT string
  Method    SigningMethod                 - - - - - - -  signing algo (HS256, RS256, etc)
  Header    map[string]interface{}    - - - - - - -  decoded header JSON
  Claims    Claims                        - - - - - - -  decoded claims (payload)
  Signature string                         - - - - - - -  signature segment
  Valid     bool                           - - - - - - -  token validation status
}
func (t *Token) SignedString(key any) (string, error)

            * - - - - - - Any type implementing this can be used as Claims - - - - - - *
type Claims interface {
          GetExpirationTime() (*NumericDate, error)
          GetIssuedAt() (*NumericDate, error)
          GetNotBefore() (*NumericDate, error)
          GetIssuer() (string, error)
          GetSubject() (string, error)
          GetAudience() (ClaimStrings, error)
}
                                    * - - - - - - Functions you'll use - - - - - - *
func New(method SigningMethod, opts ...TokenOption) *Token
func NewWithClaims(method SigningMethod, claims Claims, opts ...TokenOption) *Token
func Parse(tokenString string, keyFunc Keyfunc, options ...ParserOption) (*Token, error)
func ParseWithClaims(tokenString string, claims Claims, keyFunc Keyfunc, options
...ParserOption) (*Token, error)

                          * - - - - - - Built in Types to support Claims- - - - - - *
type RegisteredClaims struct { ... }          <- -
type MapClaims map[string]any                 <- - - We'll learn in deep about these two
```

Token :

⇒ JWT itself is just a string:

→ “header.playload.signature”

⇒ So, Token is the **parsed version of jwt token**

⇒ Used by the library to:

→ hold header

→ hold claims

→ hold signing method

→ track validity

token.SignedString(key) :

⇒ Converts a `Token` object into a pure JWT string

⇒ **hashes** header + payload **using the given secret key**

⇒ Produces:

→ “header.payload.signature”

⇒ This is the final step of token token creation

New(method, options) :

⇒ Creates an empty jwt `Token` object without claims

⇒ Not recommended for real-world usage

⇒ If you see it’s source code, it just uses `NewWithClaims` but without Claims

NewWithClaims(method, claims, options) :

⇒ Creates jwt `Token` object with given Claims (MapClaims)

⇒ Sets: header + claims + signing method

⇒ NOTE:

→ It’s **still not signed**

→ Signing always happens later using `SignedString()`

MapClaims :

⇒ we’ll learn about this in the coming section ‘16.1.2. Deep Dive into Claims’

RegisteredClaims :

⇒ we’ll learn about this in the coming section “16.1.2. Deep Dive into Claims”

Parse(tokenString, keyFunc, options) :

⇒ Parses the raw JWT string into a Token object.
⇒ Validates signature, signing method, token validity
⇒ Sets `token.Valid = true` only when signature and validations passes
⇒ Populates `token.Claims` using the **default** `MapClaims` type
⇒ Returns the parsed Token or an error.

ParseWithClaims(tokenString, claims, keyFunc, options) :

⇒ Same as `Parse()` but lets you supply your own claims type
→ (usually a struct like custom or RegisteredClaims).
⇒ Used for Type safety

16.1.2. Deep Dive into Claims

⇒ We know, Claims are the JSON payload of JWT.

⇒ We have seen `Claims` as interface in `Token`
⇒ So any type which satisfies that interface can be used as Claims

⇒ Two ways to create claims:
→ **RegisteredClaims (struct)** — standard JWT fields built-in
→ **MapClaims (map[string]any)** — flexible but no validation, not type-safe
⇒ Both satisfy the `Claims` interface.

1. Registered Claims

⇒ They are standard, well-known JWT fields
⇒ Examples: iss (issuer), sub (subject / userid), aud(audience), etc...

```
type RegisteredClaims struct {
        Issuer string `json:"iss,omitempty"`
        Subject string `json:"sub,omitempty"`
        Audience ClaimStrings `json:"aud,omitempty"`
        ExpiresAt *NumericDate `json:"exp,omitempty"`
        NotBefore *NumericDate `json:"nbf,omitempty"`
        IssuedAt *NumericDate `json:"iat,omitempty"`
        ID string `json:"jti,omitempty"`
}

func (c RegisteredClaims) GetExpirationTime() (*NumericDate, error)
func (c RegisteredClaims) GetNotBefore() (*NumericDate, error)
func (c RegisteredClaims) GetIssuedAt() (*NumericDate, error)
func (c RegisteredClaims) GetAudience() (ClaimStrings, error)
func (c RegisteredClaims) GetIssuer() (string, error)
func (c RegisteredClaims) GetSubject() (string, error)
```

⇒ **To add custom claims, embed RegisteredClaims inside your struct.**
(We'll see example letter)

2. Map Claims

⇒ It's just a Named-Type of map which satisfies the `Claims` interface

⇒ **Down Point**

→ No type safe

⇒ it's used by default but not recommended

```
type MapClaims map[string]any

func (m MapClaims) GetExpirationTime() (*NumericDate, error) {}
func (m MapClaims) GetNotBefore() (*NumericDate, error) {}
func (m MapClaims) GetIssuedAt() (*NumericDate, error) {}
func (m MapClaims) GetAudience() (ClaimStrings, error) {}
func (m MapClaims) GetIssuer() (string, error) {}
func (m MapClaims) GetSubject() (string, error) {}
...
```

A real example of how you create claims

⇒ Example on Struct

⇒ We embed the RegisteredClaims:

→ Gives us StandardClaims too

→ Gives us access to `Claims` interface too

```
type AuthClaims struct {
        UserID string `json:"sub"`
        Role   string `json:"role"`
        jwt.RegisteredClaims
}
```

⇒ It's used with:

→ **NewWithClaims()** → To create new JWT Token Object

→ **ParseWithClaims()** → To Parse token to JWT Token Object

16.1.3. JWT Creation Flow(Code)

⇒ To Create JWT we follow 5 steps:

→ Step 1: Choose Signing Method

→ Step 2: Define Claims

→ Step 3 : Create Claims Value

→ Step 4: Create Token Object

→ Step 5: Sign the Token

Step 1: Choose Signing Method :

⇒ JWT must declare how it's signed

⇒ For example we'll use `HS256`

```
method := jwt.SigningMethodHS256
```

Step 2: Define Claims :

```
type AuthClaims struct {
    UserId string `json:"userid"`
    Role string `json:"role"`
   jwt.RegisteredClaims      - - - - - - - - - - - Embedding RegisteredClaims
}
```

⇒ sub(subject) = user identifier

⇒ role = custom claim

⇒ RegisteredClaims (composited) → handles validation for registered claims

Step 3 : Create Claims Value :

```
claims := AuthClaims{
        UserID: "user_123",
        Role:   "admin",
        RegisteredClaims: jwt.RegisteredClaims{
                ExpiresAt: jwt.NewNumericDate(time.Now().Add(15 * time.Minute)),
                IssuedAt:  jwt.NewNumericDate(time.Now()),
        },
}
```

Step 4: Create Token Object :

```
token := jwt.NewWithClaims(method, claims)
```

⇒ Token now has:

→ header, claims and signing method

⇒ **REMEMBER**: It's not signed yet

Step 5: Sign the Token :

```
secretKey := []byte("your-super-secret-key")

jwtString, err := token.SignedString(secretKey)
if err != nil {
        log.Fatal(err)
}
```

16.1.4. JWT Verification Flow(Code)

⇒ To validate the jwt token we follow just 3 steps

→ Step 1: Parse token with TypedClaims

→ Step 2: Check Token Validity

→ Step 3 : UseClaims

Step1 : Parse with Typed Claims :

```
parshedClaims := &AuthClaims{}

keyFunc := func(token *jwt.Token) (any, error) {
        return []byte("your-super-secret-key"), nil
}

token, err := jwt.ParseWithClaims(jwtString, parsedClaims, keyFunc)
```

```
if err != nil {
        panic(err)
}
```

⇒ NOTE: when you use `ParseWithClaims` it will populate the struct that you gave (for eg: `&parshedClaims` here) now your claims will be in `parshedClaims` and your `token.Claims` will have pointer to `parshedClaims`.

→ if you don't pass the pointer then it will store the entire copy(not pointer) in the token object

⇒ Remember: you can use `Parse()` too but that will put claims as `MapClaim` in `token.Claims`

→ remember ? It satisfies the Claim interface too

Step 2: Check Token Validity :

```
if !token.Valid {
    panic("Token is not valid")
}
```

Step3: UseClaims :

```
fmt.Println(parshedClaims.UserID)
fmt.Println(parshedClaims.Role)
```

16.1.5. Overall Example

NOTE: This is raw non-production code just for the grab of it

```
type TheClaims struct {
        UserId string `json:"user_id"`
        jwt.RegisteredClaims
}

func main() {
        jwtString := CreateToken()
        ValidateToken(jwtString)
}

func CreateToken() string {
        claims := TheClaims{
            UserId: userId,
            RegisteredClaims: jwt.RegisteredClaims{
                ExpiresAt: jwt.NewNumericDate(time.Now().Add(time.Hour)),
                IssuedAt:  jwt.NewNumericDate(time.Now()),
                Issuer:    "your-app",
                Subject:   userId,
            },
        }

        signingMethod := jwt.SigningMethodHS256
        tokenObject := jwt.NewWithClaims(signingMethod, claims)
        signedString, err := tokenObject.SignedString([]byte("your-secret-key"))
        if err != nil {
                fmt.Println("Error occurred while signing string")
                panic(err)
        }
        fmt.Println("The signed string is: \n", signedString)
        return signedString
}

func ValidateToken(token string) {
        claims := TheClaims{}
```

```
	getKeyFunc := func(*jwt.Token) (any, error) {
		return []byte("your-secret-key"), nil
	}

	jwt, err := jwt.ParseWithClaims(token, &claims, getKeyFunc)
	if err != nil {
		fmt.Println("Error occurred while parsing Claims")
	}

	if !jwt.Valid {
		fmt.Println("Invalid Token!")
	}

	fmt.Printf("\nThe jwt token obj after parsing: %#v", jwt)
	fmt.Printf("\n\nThe claims after parsing: %#v", claims)
}
```

16.2. OAuth Authentication

16.3. *** We'll continue on this part on future version of the Book***

17. Logging

⇒ Assuming you already have basic understanding of Logging

17.1. Introduction

⇒ Logging is not printing
⇒ Logging is structured, intentional recording of system state

⇒ This is **not Logging**:

```
User logged in
```

⇒ This is **Logging**:

```
2026-01-23T10:12:04Z INFO request_completed method=GET path=/login status=200 latency_ms=42
```

⇒ And this is **Good Logging**:

```
{
  "timestamp": "2026-01-23T10:12:04Z",
  "level": "INFO",
  "message": "request_completed",
  "method": "GET",
  "path": "/login",
  "status": 200,
  "latency_ms": 42
}
```

17.2. Logging Options in Go

In Go there are many ways to Log:

1. fmt.Println() :

⇒ Prints text. No levels, no timestamps. Debug only.
⇒ It's logging too, but DO NOT USE IT

2. `log` package :

⇒ Basic logger. Adds timestamp. Plain text.
⇒ Built-in in Go
⇒ Not Structured

3. `slog` package :

⇒ Built-in structured logging
⇒ Better than log
⇒ Still limited compared to advanced external packages

⇒ External package loggers are way more advanced and faster than built in

4. `zerolog` package :

⇒ Always structured JSON.
⇒ Zero-allocation
⇒ Extremely fast

5. `zap` package :

⇒ Started by uber
⇒ Fast, structured, production-grade
⇒ Slightly heavier than zerolog

⇒ We'll be learning `zerolog`

17.3. Introduction to `zerolog`

⇒ `zerolog` is an external structured logging library for Go
⇒ JSON-first design
⇒ No pretty output by default (good)

⇒ Pretty console available via `zerolog.ConsoleWriter`
→ Use only in development (inefficient)

Specially:
⇒ Zero-allocation logging
⇒ No Reflection

Install it:

```
go get -u github.com/rs/zerolog/log
```

17.4. Core structure of zerolog

```
type Event struct {
        ...
}
type Logger struct {
        ...
}
type Level uint8
const (
   PanicLevel Level = iota
   FatalLevel
   ErrorLevel
   WarnLevel
   InfoLevel
   DebugLevel
   TraceLevel
)
      * - - - - - - - - - - - - - Event Methods you'll use (chainable) - - - - - - - - - - - - -  *
func (l Logger) Info() *Event
func (l Logger) Debug() *Event
func (l Logger) Error() *Event
func (l Logger) Warn() *Event
func (l Logger) Fatal() *Event
func (l Logger) Panic() *Event
```

```
func (e *Event) Str(key, val string) *Event
func (e *Event) Int(key string, val int) *Event
func (e *Event) Float64(key string, val float64) *Event
func (e *Event) Bool(key string, val bool) *Event
func (e *Event) Time(key string, val time.Time) *Event
func (e *Event) Err(err error) *Event
func (e *Event) Msg(msg string) error
func (e *Event) Msgf(format string, args ...any) error
func (e *Event) Stack() *Event
...

            * - - - - - - - - - - - - - Logger Methods you'll use - - - - - - - - - - - - - *
func New(w io.Writer) Logger

func (l Logger) With() Context
func (l Logger) Level(level Level) Logger
func (l Logger) Output(w io.Writer) Logger
func (l Logger) WithContext(ctx context.Context) context.Context

type Context struct {
   logger Logger
}
func (c Context) Timestamp() Context
func (c Context) Logger() Logger

                * - - - - - - - - - - - - - Global Functions - - - - - - - - - - - - - *
func SetGlobalLevel(level Level)
func MultiLevelWriter(writers ...io.Writer) io.Writer     - - - - - - - - - ⇒ multi output helper

                * - - - - - - - - - - - - - Pretty Output- - - - - - - - - - - - - *
type ConsoleWriter struct {
   Out        io.Writer
   NoColor    bool
   TimeFormat string
}
                        * - - - - - - - - - - - - - Context - - - - - - - - - - - - - *
func Ctx(ctx context.Context) *Logger        // Attach stack trac
```

17.5. Simple Logging (Global Logger)

```
import “github.com/rs/zerolog/log”

func main () {
    log.Info().Msg(“Hello World”)
    log.Error().Err(err).Msg("Got Error")

    log.Info().Send()
}
```

Output:

```
{"level":"info","time":"2026-01-27T08:22:54+05:45","message":"Hello World"}
{"level":"error", “error”:”Error_message_here”,"time":"2026-01-27T08:22:54+05:45",
"message":"Got Error"}
```

BigNOTE

Msg()/Send() MUST be in the end of the chain
other wise it won’t perform the log

17.6. Structured Fields

⇒ `zerolog` calls them **context** too

```
log.Debug().Str(“user”, “gopher”).Int(“age”, 22).Msg(“Got Profile”)
```

Result:

```
{"level":"debug","user":"gopher","age":22,"time":"2026-01-27T08:18:51+05:45","message":"Got
Profile"}
```

17.7. Custom Logger (Deep Dive)

⇒ Creating a custom logger instance is a whole separate section on it self, so hold up your focus and try to understand everything.

17.7.1. Core Building Block

⇒ Let's have a quick look on the architecture we'll be using to create a new Logger

```
// Main Logger
type Logger struct {
        ...                    < - - - - - - - - - - - - internal fields
}
func New(w io.Writer) Logger
func (l Logger) With() Context
func (l Logger) Level(lvl Level) Logger
func (l Logger) Sample(s Sampler) Logger
...

type Context struct {     < - - - - plays a crucial role (we'll learn about it deeply in next page)
        l      Logger
}
func (c Context) Timestamp() Context
func (c Context) Str(key, val string) Context
func (c Context) Int(key string, i int) Context
func (c Context) Logger() Logger
...

type Event struct {
        ...
}
func (e *Event) Int(key string, i int) *Event
func (e *Event) Str(key, val string) *Event
func (e *Event) Msg(msg string)
...
```

Big Note	We have many more other methods in the similar way

17.7.2. [Example]: Creating custom logger

```
baseLogger:= zerolog.New(os.Stdout)
newLogger := baseLogger.With().Timestamp().Str("user_id", "User123").Logger()

newLogger.Info().Int("status_code", 200).Msg("User logged in successfully")
```

- - - - ⇒ **I know this looks confusing. Now, we'll understand it deeply** ⇐ - - - -

17.7.3. [Critical]: How Logger Creation Actually Works

First, understand few things:

⇒ `Loggers` are immutable

→ it's configurations never changes

→ Because logging happens millions of times, and configuration should not.

⇒ `Context` is a **builder for fields**, not a logger.

⇒ `Event` triggers the actual logging.

⇒ Since Loggers are immutable, to change the logger :

→ You must create new logger one out of it

Now, Let's understand Step-by-step:

⇒ **zerolog.New()** → creates **base logger with Writer you gave**

⇒ **.With()** → creates new Context builder from base logger.

⇒ **.Timestamp()** → adds default **'time'** field to Context

⇒ **.Str()** → adds given default '**key and value'** field to Context

Finally,

⇒ **.Logger()** → creates a new immutable Logger with Context's config.

17.7.4. Loggers Hierarchy

⇒ Now that you understand how logger creation works, you now realise that:

→ You can create as many new loggers as you want from any existing logger

```
logger1 := zerolog.New(os.Stdout).With().Timestamp().Str("service", "api").Logger()
logger2 := logger1.With().Str("request_id", "abc123").Logger()
logger3 := logger2.With().Str("user_id", "user456").Logger()
```

⇒ Each new logger inherits all fields from its parent, plus new defaults you add. No mutation anywhere.

17.8. Writing Log into a File

Core Idea:

⇒ zerolog does NOT know about files, consoles, or terminals
⇒ zerolog only knows about `io.Writer`
⇒ that io.Writer can be “FileWriter” or “ConsoleWriter” or whatever.

Rule:

⇒ If something implements `io.Writer`, zerolog can writer logs to it

So:

⇒ Writing logs to file = give a file (which is `io.Writer`) to the logger

Core Function We’ll Use

⇒ We saw this method in the previous chapter `17.7. Custom Logger (Deep Dive)` too

```
func New(w io.Writer) Logger
```

⇒ Returns new Logger which writes to the `w io.Writer`

Step 1 : Create a File Writer

```
file, err := os.OpenFile(
    "app.log",
    os.O_CREATE|os.O_WRONLY|os.O_APPEND,
    0644,
)
if err != nil {...}
```

Step 2: Create new logger using that writer

```
logger := zerolog.New(file).With().Timestamp().Logger()
```

Result:

⇒ Every log event is written into app.log
⇒ No difference from console logging
⇒ Only the writer changed

17.9. Concept of “Log Levels” in zerolog

17.9.1. Introduction

⇒ Log levels control which log messages get output and which get ignored.
⇒ They help manage noise by filtering logs based on importance or environment.

17.9.2. Basic Rule

If **log level < configured level** → **ignored**
If **log level ≥ configured level** → **written**

17.9.3. Log Levels

Panic
↑
Fatal
↑
Error
↑
Warn
↑
Info
↑
Debug
↑
Trace

Example:
⇒ If you set configuration level to **Info**
⇒ then **Debug**, **Trace** are → **ignored**

- - - - - - ⇒ Let's see the code examples now ⇐ - - - - - -

17.9.4. Log Levels in global Logger

```
zerolog.SetGlobalLevel(zerolog.InfoLevel)

log.Info().Msg("This will show")          - - - - - Shows✅
log.Trace().Msg("This will not show")     - - - - - Ignored❌
```

17.9.5. Log Levels in custom Logger

```
logger := zerolog.New(os.Stdout).Level(zerolog.InfoLevel).With().Timestamp().Logger()

logger.Info().Msg("This will show")          - - - - - Shows✅
logger.Trace().Msg("This will not show")     - - - - - Ignored❌
```

You might ask Why does .Level() come before .With()?

⇒ .Level() is method of Logger not Context

⇒ With() returns Context so you can't call it from there.

⇒ It doesn't come with Context because:

→ It is Logger's behaviour

→ Context is for contexts(fields coming in logger)

⇒ Same as doing:

```
baseLogger:= zerolog.New(os.Stdout)
logger2 := baseLogger.Level(zerolog.InfoLevel)
logger3 := With().Timestamp().Logger()
```

17.10. Sampling

17.10.1. Introduction

⇒ Sampling means logging only a subset of log events
⇒ Used when logs are high-frequency and repetitive
⇒ **In simple terms:**
→ Sampling is like intentionally skipping logs in a controlled and predictable way

17.10.2. Using Sampling with Custom Logger

```
logger := zerolog.New(os.Stdout).With().Timestamp().Logger()

sampled := logger.Sample(&zerolog.BasicSampler{N: 3})

sampler.Info().Msg("Will Show")    - - - - - - - - - ✅
sampler.Info().Msg("Won't Show")   - - - - - - - - - ❌
sampler.Info().Msg("Won't Show")   - - - - - - - - - ❌
sampler.Info().Msg("Will Show")    - - - - - - - - - ✅
```

⇒ **Meaning:** Log 1 out of every 3 log calls

17.10.3. Types of Samplers in Zerolog

⇒ There are few types of Samplers

17.10.4. Basic Sampler

⇒ Log 1 out of N events

```
&zerolog.BasicSampler{N: 10}
```

17.10.5. Burst Sampler

⇒ Allows a burst, then throttles

```
&zerolog.BurstSampler{
    Burst: 3,
    Period: time.Second * 3,
}
```

⇒ First 3 logs pass
⇒ Remaining logs dropped for 3 seconds

17.10.6. Level Sampler

⇒ Different sampling per log level

```
&zerolog.LevelSampler{
    InfoSampler:  &zerolog.BasicSampler{N: 10},
    DebugSampler: &zerolog.BasicSampler{N: 50},
}
```

⇒ Info logs sampled 1 out of 10
⇒ Debug logs sampled 1 out of 50

17.11. Multi Outputs

⇒ If you want the same log event sent to multiple places simultaneously, zerolog makes it simple.

Remember: zerolog writes to anything implementing io.Writer (section '17.8. Writing Log into a File')

How it works:

⇒ To write to a file, you give "FileWriter" to Logger, right?

⇒ So, to write into multiple places, you give a "Writer" which writes in multiple places
⇒ That's where zerolog.MultiLevelWriter comes in.

Core Structure

```
func MultiLevelWriter(writers ...io.Writer) io.Writer
```

⇒ This takes multiple `io.Writer`s
⇒ Returns a single io.Writer that writes to all of them
⇒ Use this to create a new logger with multiple outputs

Example

```
// Output 1 (Console Writer)
consoleWriter := zerolog.ConsoleWriter{Out: os.Stdout}

// Output 2 (File Writer)
fileWriter, err := os.Create("app.log")
if err!= nil {
    panic(err)
}

// Create logger with multi output
multi := zerolog.MultiLevelWriter(consoleWriter, fileWriter)

// Now you create a new logger giving this multi writer to it (you know it)
logger := zerolog.New(multi).With().Timestamp().Logger()

logger.Info().Str("userid", "userid").Send()
  ↑ - - - - - - You use it normally and it will write in multiple outputs
```

17.12. Logger integration with context.Context

⇒ In Go backend, you pass loggers via request context

→ For things like: unique Trace Ids, default http Method field, etc

⇒ You can manually put loggers in Context and retrieve them

→ But `zerolog` offers a cleaner, easier way

Core Structure of Helper Functions

```
type Logger struct {
...
}
func (l Logger) WithContext(ctx context.Context) context.Context {...}


func Ctx(ctx context.Context) *Logger {...}
```

logger.WithContext(ctx) :

⇒ Takes context

⇒ Gives a new derived Context with `Logger` as Value

Ctx(ctx) :

⇒ Takes context

⇒ Takes out your Logger by using that its key

Code Example

```
rootContext := context.Background()
logger := zerolog.New(os.Stdout).Timestamp().Logger()

ctx := logger.WithContext(rootContext)
...

theLogger := zerolog.Ctx(ctx)
theLogger.Info().Msg("logger from context")
```

It's just like doing

```
ctx := context.WithValue(rootContext, "logger", logger)    - - - - put logger in context

loggerFromCtx := ctx.Value("logger").(zerolog.Logger)    - - - - take out logger from context
loggerFromCtx.Info().Msg("logger from context")
```

17.13. Error and Stack

Basic Error Logging :

```
err := errors.New("oops")

logger.Error().Err(err).Msg("something bad")
```

⇒ But basic errors don't show where they happened.
⇒ So, we need to have the **"Stack Traces"** too

"Stack Traces" = list of function calls leading to the error, showing exactly where it occurred.

⇒ For Stack Trace we use the external package called → **`errors`**
install package→ `go get github.com/pkg/errors`

Advanced Error Logging :

```
zerolog.ErrorStackMarshaler = errors.MarshalStack

logger.Error().Stack().Err(err).Msg("error with stack trace")
```

Output JSON includes ⇒ **"stacktrace"** field with a call stack.

Output will be something like this:

```
{
  "level": "error",
  "error": "oops",

"stacktrace":"main.doSomething\n\t/path/main.go:25\nmain.main\n\t/path/main.go:15 \nruntime.main\n\t/usr/local/go/src/runtime/proc.go:225",
  "message": "error with stack trace",
  "time": "2026-01-29T10:00:00Z"
}
```

18. Go as Client

18.1. Introduction

⇒ Every server talks to other servers
⇒ For that, a server must act as an HTTP Client
⇒ For that, we should know how to use Go as an HTTP Client

⇒ The same `net/http` which was helping us build **Server**
⇒ Also helps us with make HTTP request as a **Client**

18.2. Introduction to `http.Client` and `http.Request`

⇒ To make HTTP requests as a client, `net/http` gives you two core types:

1. http.Client :

⇒ `http.Client` is a sender
⇒ It knows how to talk to the network
⇒ Handles connection reuse, TLS, timeouts, proxies

⇒ Just like `http.Server` is for receiving request, `http.Client` is for sending request
⇒ You don't build raw TCP connections; you ask `http.Client` to do it for you
⇒ Think of it as the **Browser in FrontEnd**

2. http.Request :

In Client-Side(Outgoing):
⇒ Request you build and send with help of `http.Client
⇒ Contains method, URL, headers, body
⇒ Same idea as `fetch()` / `axios` in FrontEnd
→ They build request and send with the help of browser

⇒ The same `http.Request` that we learned in Server-Side too
→ check '5. http.Request Anatomy'

In Layman

http.Request = What to send (the full detailed HTTP message)
http.Client = How to send it (connection, timeout, redirects, cookies) (like a browser)

⇒ Check the section '18.11. Final Mental Model' if you already want to get the full picture view

18.3. Core structure of `http.Client`

```
type Client struct {
        Transport RoundTripper
        CheckRedirect func(req *Request, via []*Request) error
        Jar CookieJar
        Timeout time.Duration
}

func (c *Client) Get(url string) (resp *Response, err error)
func (c *Client) Do(req *Request) (*Response, error)
func (c *Client) Post(url, contentType string, body io.Reader) (resp *Response, err error)
func (c *Client) PostForm(url string, data url.Values) (resp *Response, err error)
```

⇒ Core structure of `**http.Request**` in section '5. http.Request Anatomy'
⇒ Core structure of `**http.Response**` in section '18.4. Core structure of `http.Response` For Client-Side'

type Client struct :
⇒ The HTTP client that sends http request and gets responses
⇒ Just like `**http.Server**` is for receiving request, `**http.Client**` is for sending request
⇒ Uses the same structs `**http.Request**` and `**http.Response**`

Client.Transport :
⇒ Controls how requests are sent over the network
⇒ Handles connections, proxies, TLS
⇒ Default is fine for most cases

Client.CheckRedirect :
⇒ Function that controls what happens when the server sends redirects
⇒ Called when server returns **3xx redirect**
⇒ Lets you allow, block, or limit redirects
⇒ Prevents redirect loops

Client.Jar :
⇒ Manages cookies automatically
⇒ Useful for session-based requests
⇒ If nil → cookies are ignored

Client.Timeout :
⇒ Maximum time that Client waits for a response
⇒ Stops hanging requests after this duration
⇒ Always set this in real systems

Do(Request) (*Response, error) :
⇒ Core method of `http.Client` to send request over the network
⇒ It's the core method behind all other shortcuts

⇒ Takes **`http.Request`** you build
⇒ Sends it over the network using **`http.Client`**
⇒ Returns the **`*http.Response`** or **`error`**

Get(url) (*Response, error) :
⇒ **Shortcut** for sending a **simple GET request** to the given url
⇒ Builds **`http.Request`** behind the scenes
⇒ Than calls **`Do()`**

Post(url, contentType, body) (*Response, error) :
⇒ **Shortcut** for sending a **simple POST request** to the Url
⇒ Builds **`http.Request`** behind the scenes
⇒ Than calls **`Do()`**

PostForm(url, data) (*Response, error) :
⇒ **Shortcut** for sending a **POST request with form data**
⇒ Takes form values as **`url.Values`** and encodes automatically
⇒ Sets content-type to "application/x-www-form-urlencoded"
⇒ Calls **`Post()`** method

18.4. Core Structure of `http.Response` (For Client-Side)

NOTE

⇒ We're learning the core structure of it for Client-Side
⇒ We've learned about it for Server-Side in section
→'5.3. Core Structure of `http.Request` (For Server-Side)'

```
                    * - - - - - - - - - http.Request  - - - - - - - - - *
type Request struct {
   Method             string
   URL                *url.URL
   Proto              string
   ProtoMajor         int
   ProtoMinor         int
   Header             Header
   Body               io.ReadCloser
   GetBody            func() (io.ReadCloser, error)
   ContentLength      int64
   TransferEncoding   []string
   Host               string
   Trailer            Header
}
⇒ Other fields exists too, but used internally or in Server-Side

                    * - - - - - - - - - Methods  - - - - - - - - - *
func (r *Request) WithContext(ctx context.Context) *Request
func (r *Request) Clone(ctx context.Context) *Request
func (r *Request) AddCookie(c *Cookie)
func (r *Request) Write(w io.Writer) error
⇒ Other methods exists too, but used internally or in Server-Side

                    * - - - - - - - - - Functions  - - - - - - - - - *
func NewRequest(method, url string, body io.Reader) (*Request, error)
func NewRequestWithContext(ctx context.Context, method, url string, body io.Reader)
(*Request, error)
```

type Request struct :
⇒ Represents an HTTP request
⇒ In Client-Side → Created by you
⇒ In Server-Side → Auto populated by **`net/http`**

*- - - - - - - - - - Fields - - - - - - - - - - *

Request.Method :
⇒ HTTP method (GET, POST, etc)

Request.URL :
⇒ Parshed request URL
⇒ Scheme, Host, Path, Query, etc

Request.Proto :
⇒ HTTP protocol versions has string
⇒ eg: “HTTP/1.1”
⇒ mainly used in Server-Side or internally
⇒ Just here for you to know it exists

Request.ProtoMajo & **Request.ProtoMinor :**
⇒ Proto version in integer
⇒ mostly internal use
⇒ Just here for you to know it exits

Request.Header :
⇒ Map of HTTP Headers for request metadata

Request.Body :
⇒ Request body as `io.ReadCloser`

Request.GetBody :
⇒ Optional func place to recreate Body
⇒ For advanced customization
⇒ Rarely used in Client-Side... Just here for you to know it exists

Request.ContentLength :
⇒ Length of Body (in bytes) (-1 if unknown)

Request.TransferEncoding :
⇒ Transfer encoding list (eg. chunked)
⇒ Mostly Used Internally... Just here for you to know it exsits

Request.Host :
⇒ Host header override (eg. custom.example.com)

Request.Trailer :
⇒ Trailer Headers (rarely used)
⇒ Trailers are → HTTP headers that will be sent after the body

** - - - - - - - - - Methods - - - - - - - - - **

WithContext(ctx) *Request :
⇒ Returns new shallow copy of Request with new context

Clone(ctx) *Request :
⇒ Returns deep copy of Request with new context

AddCookie(c) :
⇒ Add a cookie header easily

Write(w) :
⇒ This Serialize requests to bytes
⇒ You don't use it directly... Just here for you to know it exists

** - - - - - - - - - Creation Functions - - - - - - - - - **

NewRequestWithContext(ctx, method, url, body) (*Request, error) :
⇒ Builds and returns new `http.Request`
→ with given ctx, method, url and body

NewRequest(method, url, body) (*Request, error) :
⇒ It's just calling `NewRequestWithContext`
→ but context.Background() as a context

18.5. Core Structure of `http.Response`

```
type Response struct {
        Status              string
        StatusCode          int
        Proto               string
        ProtoMajor          int
        ProtoMinor          int
        Header              Header
        Body                io.ReadCloser
        ContentLength       int64
        TransferEncoding    []string
        Close               bool
        Uncompressed        bool
        Trailer                     Header
        Request             *Request
        TLS                 *tls.ConnectionState
}

func (r *Response) Cookies() []*Cookie
⇒ Other methods exists too, but for internal use only
```

⇒ See the core structure of **Header** in section '10.5. Core Structure of Header'
⇒ See the core structure of **Cookie** in section '13.1. Core structure of Cookie'

Big NOTE

⇒ If you do Front-End
⇒ Just visualize `http.Response` as response you get from API in Frontend

type Response struct :
⇒ Represents parsed structured version HTTP response
⇒ Auto populated by `net/http` → `Client`
⇒ You only read from it

Response.Status :
⇒ HTTP status line as string (eg: "200 OK")

Response.StatusCode :
⇒ Status Code in Integer type (eg: 200, 400)

Response.Proto, **ProtoMajor**, **ProtoMinor**, **Header**, **Body**, **ContentLength**, **TransferEncoding**, **Close**, **Uncompressed**, **Trailer**, **TLS** :
⇒ Every field is same as **`http.Request`** in Client-Side
⇒ Just carries metadata of what's incoming

Response.Request :
⇒ The pointer to the `http.Request` that triggered this response

Response.Body :
⇒ Same as `http.Request.Body` but it's response body
⇒ **Streamed** through another server to our "Client-Side" Go

** - - - - - - - - - - Method - - - - - - - - - - **

Cookies() []*Cookie :
⇒ Same as you'd use Cookies from **`http.Request`**
⇒ Learn more about it in section '13. Cookies'

18.6. [Example]: Creating a client

```
                    * - - - - - -  Basic Creation Example (Most Cases)  - - - - - -*
client := &http.Client{
   Timeout: 5 * time.Second,
}

                                * - - - - - -  Advanced Example - - - - - -*
client := &http.Client{
   Timeout: 10 * time.Second,
   Transport: &http.Transport{
      MaxIdleConns:       10,
      IdleConnTimeout:    30 * time.Second,
      TLSHandshakeTimeout: 5 * time.Second,
      DisableKeepAlives:   false,
      Proxy:              http.ProxyFromEnvironment,
   },
   CheckRedirect: func(req *http.Request, via []*http.Request) error {
      if len(via) >= 5 {
         return http.ErrUseLastResponse
      }
      return nil
   },
   Jar: cookieJar,     - - - - - -> cookieJar implements http.CookieJar interface
}
```

18.7. [Example]: Creating a request

```
ctx := context.Background()
method := http.MethodPost
url := "https://api.example.com/users"
body := bytes.NewBuffer([]byte(`{"username":"Diwash","password":"secret123"}`))

var req *http.Request
var err error

req, err = http.NewRequestWithContext(ctx, method, url, body) - - - > or NewRequest()
if err != nil {
    ...
}

req.Header.Set("Content-Type", "application/json")

req.Host = “api.example.com”       - - - - - - - > optional override

req.AddCookie(&http.Cookie{Name: "session", Value: "abc123"})
```

18.8. [Example]: Sending a Request with Client

Using `client.Do(req)`

```
                * - - - - - -  Build Request (like previous example) - - - - - -*
method := http.MethodPost
url := "https://api.example.com/users"
body := bytes.NewBuffer([]byte(`{"username":"Diwash","password":"secret123"}`))

req, err := http.NewRequest(method, url, body)
if err != nil {...}

                * - - - - - -  Calling Do() - - - - - -*

var resp *http.Response
resp, err = client.Do(req)
if err != nil {...}
defer resp.Body.Close()
```

Using `client.Get(req)`

```
var resp *http.Response
var err error

resp, err = client.Get("https://api.example.com/data")
if err != nil {...}
defer resp.Body.Close()
```

Using `client.Post(req)`

```
var resp *http.Response
var err error

var body io.Reader = strings.NewReader(`{"name":"Diwash"}`)
url := "https://api.example.com/users"

resp, err = client.Post(url, "application/json", body)
if err != nil {...}
defer resp.Body.Close()
```

18.9. [Example]: Reading the Response coming from Request

NOTE AGAIN

⇒ `http.Response` in Client-Side is very similar to `http.Request` in Server-Side

and

⇒ `http.Response` in Client-Side
→ It's a response coming from another server
→ You don't modify it
→Same mental Model as reading an API response in frontend

```
* - - - - - - Sending the request first - - - - - - *
var err error
var resp *http.Response
resp, err = client.Do(req)
if err != nil {...}
defer resp.Body.Close()

* - - - - - - Now comes the real Response- - - - - - *

fmt.Println("Status Code:", resp.StatusCode)
fmt.Println("Status:", resp.Status)
fmt.Println("Headers:", resp.Header)
fmt.Println("Content-Length:", resp.ContentLength)
fmt.Println("Was Uncompressed:", resp.Uncompressed)

* - - - - - - Reading Body from Response - - - - - - *

data, err := io.ReadAll(resp.Body)        - - - - - - > or use json.NewDecoder()
if err != nil {...}
```

⇒ Learn more about **Body** in section '11. Body'

18.10. http.Get(), http.Post() & http.PostForm()

```
// http.Get()
func Get(url string) (resp *Response, err error)

// http.Post()
func Post(url, contentType string, body io.Reader) (resp *Response, err error)

// http.PostForm()
func PostForm(url string, data url.Values) (resp *Response, err error)
```

⇒ These are → simple helpers **for Quick Requests**

⇒ Use when you simply want to hit an API
⇒ You don't want to build Client
⇒ You don't want to build Request
⇒ It does every build ups for you
⇒ Uses `default client` + `default request`

Get(url) (*Response, err) :
⇒ Takes URL
⇒ Builds basic `http.Request`
⇒ Uses default `http.Client`
⇒ Sends request and returns response

Post(url, contentType, body) (*Response, err) :
⇒ Takes URL, content type, and body
⇒ Sets Content-Type header
⇒ Builds basic `http.Request`
⇒ Uses default `http.Client`
⇒ Sends request and returns response

PostForm(string, data) (*Response, err) :
⇒ Takes URL and form data
⇒ Encodes data as `application/x-www-form-urlencoded`
⇒ Sets Headers
⇒ Uses default client
⇒ Sends request and returns response

- - - ⇒ **They are just built for basic requests, not for configured requests** ⇐ - - -

18.11. Final Mental Model

`http.Client` :

⇒ **`http.Client`** is a sender
⇒ It knows how to talk to the network
⇒ Handles connection reuse, TLS, timeouts, proxies
⇒ Think of it as the **Browser in FrontEnd**

⇒ Just like **`http.Server`** is for receive request, **`http.Client`** is for sending request

`http.Request` :

In Client-Side(Outgoing):
⇒ Request you build and send with help of `http.Client`
⇒ Contains method, URL, headers, body
⇒ Same idea as **`fetch()`** / **`axios`** in FrontEnd
→ They build request and send with the help of browser

In Server-Side (Incoming):
⇒ Request you receive from a client
⇒ Same struct, different direction

`http.Response` :

In Client-Side(Incoming):
⇒ Response returned by another server
⇒ You only read it
⇒ Status code, headers, body (streamed)
⇒ Like API response in FrontEnd

In Server-Side (Outgoing):
⇒ Response you build and send
⇒ You write status, headers & body
⇒ You don't directly modify it in ServerSide
→ You do it with **`http.ResponseWriter`**

`http.Request` & `http.Response` :

They are same struct used in different direction

http.Client.Do(req) :

⇒ Core of every other Methods
⇒ You build
→ Your own Client (like browser)
→ Your own Request
→ Manually send it with `Do()`

⇒ Every other methods use this under the hood

http.Client.Get() , http.Client.Post() , http.Client.PostForm() :

⇒ You make your own Client
⇒ But request is simple
⇒ `Client` build the request for you
⇒ You just give URLs and Body

http.Get() , http.Post() , http.PostForm() :

⇒ You don't want to build Client
⇒ You don't want to build Request
⇒ It does everything for you

⇒ Uses `default client` + `default request`

19. Testing

⇒ Testing is nothing but a function that:

→ Calls other function

→ Check its output

→ And report if they passed or failed

⇒ But doing this manually is painful, so Go gives us tools to make it easier

⇒ Go provides 2 main packages for testing:

1. testing :

⇒ The core package that runs your tests and tells you if they pass or fail.

⇒ The foundation of testing in Go.

2. net/http/httptest :

⇒ Helpers specifically for testing HTTP servers.

⇒ Lets you create fake servers, requests, and responses for testing your APIs.

We'll start with the testing package to get a solid base, then move on to net/http/httptest to test HTTP stuff.

19.1. `testing` package

19.1.1. Introduction to `testing` package

⇒ `testing` is Go's built-in way to write and run tests
⇒ It helps you
→ run your test codes
→ compare results
→ and report pass/fails

19.1.2. Rules to remember while writing tests in Go

⇒ Go expects you to follow certain naming and structure rules so `go test` can find and run your tests properly.

19.1.2.1. File rules

⇒ Test files name must end with `_test.go` suffix

⇒ Example: if your code is in `helloworld.go`, your tests should be in `helloworld_test.go`
⇒ NOTE: The name `helloworld` is NOT mandatory, just the `_test.go` suffix matters.

⇒ **Why?** → Because those suffixes tell Go these files contain tests.

19.1.2.2. Test Function Signature

```
func TestXxx(t *testing.T) {...}
```

⇒ Function Name much start with → 'Test'
⇒ First Letter after 'Test' must be uppercase
⇒ Function should have single parameter `*testing.T`

⇒ Learn more about `*testing.T` in section :
→ '19.1.8. Understanding how `testing` works behind the scene'

19.1.3. Running Tests

Syntax

```
go test [options] [package...]
```

Basic Command

```
go test
```

⇒ **Runs tests in the current package (current directory).**
⇒ Finds all `_test.go` files and all functions starting with Test + capital letter.

Test other directories explicitly

⇒ For eg → if your tests modules are in tests directory

```
go test ./tests/
```

We also got some flags coming with it which helps us level up:

```
go test ./...
        ⇒ runs all tests over all directories

go test -v
        ⇒ runs test with verbose output

go test -run=TestFunctionName
        ⇒ runs tests specific test function

go test -json
        ⇒ Output test results in JSON format.

go test -timeout=second
        ⇒ Output test results in JSON format.

go test -race
        ⇒ detects data races

go test -cover
        ⇒ shows how much of your code is tested by running coverage analysis.

go test -bench
        ⇒ runs all tests over all directories

go test -failfast
        ⇒ Stop testing after the first test failure.
```

```
go test -short
        ⇒ Tell tests to skip long-running tests.
        ⇒ Your code must use `testing.Short()`

                - - - - - - - - - - - - Little More Advanced Part - - - - - - - - - - - -

go test -memprofile=filename
        ⇒ Records memory allocation profile during tests to the specified file.

go test -cpuprofile=filename
        ⇒ Records CPU usage profile during tests to the specified file.

go test -trace=filename
        ⇒ Records detailed execution traces of tests to the specified file for analysis.
```

19.1.4. Core Structure of `testing` package

```
- - - - - - - - - Main Testing Struct (Passed into every function) - - - - - - -
type T struct {
        common
        ...
}
func (t *T) Run(name string, f func(t *T)) bool {
func (t *T) Parallel()
func (t *T) Setenv(key, value string)
func (t *T) Deadline() (deadline time.Time, ok bool)

- - - - - - - - - Main helper for you (Comes with T) - - - - - - - - -
type common struct {...}
func (c *common) Log(args ...any)
func (c *common) Logf(format string, args ...any)
func (c *common) Error(args ...any)
func (c *common) Errorf(format string, args ...any)
func (c *common) Fatal(args ...any)
func (c *common) Fatalf(format string, args ...any)

func (c *common) Skip(args ...any)
func (c *common) Skipf(format string, args ...any)
func (c *common) Fail()
func (c *common) FailNow()
func (c *common) SkipNow()
func (c *common) Helper()

func (c *common) Cleanup(f func())      - - - - - - - - - - - Acts like defer for your testings

- - - - - - - - - Additional Function for level up - - - - - - - - -
func Short() bool
```

type T struct { ... }
⇒ Test Functions take this as a parameter to access all the methods

t.Run(name, function) :
⇒ Runs a **sub-test** named name with function f
⇒ Creates new `testing.T` and passes it to the given function
⇒ returns if it passed.

t.Parallel() :
⇒ Tells Go to run this test at the same time as other parallel tests
⇒ Used to make tests faster
⇒ NOTE: tests runs at sync by default if you don't use this

t.Setenv(key, string) :
⇒ Sets an environment variable only for this test
⇒ Automatically restores the old value when the test ends
⇒ Safe with parallel tests

t.Deadline() (deadline time.Time, ok bool) :
⇒ Tells you when Go will kill this test
⇒ Comes from **`go test -timeout`**=... command
⇒ Returns:
→ **time** → exact cutoff time
→ **ok** → true if timeout exists

---

type common struct { ... } :
⇒ All logging, failing, skipping comes from here
⇒ This is internal shared code
⇒ testing.T embeds (inherits) this struct

- - - - - - - - - - - - - - Base Methods - - - - - - - - - - - - - -

common.Fail() :
⇒ Marks the current test as failed
⇒ That test keeps running

common.FailNow()
⇒ Marks the current test as failed
⇒ Immediately stops execution of that particular test
⇒ **NOTE:**
→It just stops the particular test
→Other tests are NOT affected
→Kills only the goroutine tied to that **`*testing.T`**
⇒ Check section '19.1.8. Understanding how `testing` works behind the scene'

common.SkipNow() :
⇒ Skips the particular test immediately

common.Log(args) :
⇒ Prints message

common.Logf(string, args) :
⇒ Prints formatted message

- - - - - - - - - - - - - - Methods Build on top of base methods - - - - - - - - - - - - - -

common.Error(args) :
⇒ Marks test failed and prints error
⇒ That test keeps running
⇒ Equivalent to → Log() + Fail()

common.Errorf(string, args) :
⇒ Same as Error but with formatted string
⇒ Equivalent to → Logf() + Fail()

common.Fatal(args) :
⇒ Marks test failed and prints error
⇒ Immediately stops that particular test only
⇒ Equivalent to → Log() + FailNow()
⇒ **NOTE:**
→It just stops the particular test
→Other tests are NOT affected
→Kills only the goroutine tied to that `*testing.T`
⇒ Check section '19.1.8. Understanding how `testing` works behind the scene'

common.Fatalf(string, args) :
⇒ Same as Fatal() but with formatted string
⇒ Equivalent to → Logf() + FailNow()

common.Skip(args) :
⇒ Skip test with message
⇒ Equivalent to → Log() +SkipNow()

common.Skipf(string, args) :
⇒ Same as Skip() but with formatted string
⇒ Equivalent to → Logf() +SkipNow()

- - - - - - - - - - - - - - Helper Functions- - - - - - - - - - - - - -

Helper() :

⇒ Marks function it's in as helper

⇒ Error line will point to real test, not helper

Cleanup(function) :

⇒ Runs the given function after test finishes

⇒ Like defer for testing

→ Always runs even on failure

Short() bool :

⇒ Returns true if tests are run with (go test -short ...) mode

⇒ It's used for skipping some time taking tests

⇒ You run the test on -short mode and it will be true and you skip it in code.

BIG NOTE (Don't Skip)

FailNow(), Fatal() and Fatalf()

⇒ They sound like they will stop the entire test.

⇒ but they don't stop the entire test.

→ They just stop that particular test... which means TestFunction with that **`t *testing.T`**

⇒ Learn more about it in section '19.1.8. Understanding how `testing` works behind the scene'

19.1.5. Basic Code Example of Testing

⇒ Function that we'll be testing (math.go):

```
package mathutils

func Add(a, b int) int {
        return a + b
}
```

⇒ Basic Test (math_test.go)

```
package mathutils

import "testing"

func TestAdd(t *testing.T) {
        result := Add(2, 3)

        if result != 5 {
                t.Fatalf("expected 5, got %d", result)
        }
}
```

⇒ Run the test

```
go test
```

Big NOTE

⇒ A test is PASS by default
⇒ It will be marked failed when you do it

19.1.6. Problems with Normal Testing

⇒ If your test has only one case then it's okay, but

⇒ Testing multiple cases manually leads to code repetition and clutter:

```
func TestAdd(t *testing.T) {
        if Add(2, 3) != 5 {
                t.Fatal("2 + 3 failed")
        }
        if Add(-1, 1) != 0 {
                t.Fatal("-1 + 1 failed")          - - - - - - - - - Too much code repetition
        }
        // and so on...
}
```

⇒ Repetition

⇒ No clear structure

⇒ This is not how you'll be testing

- - - - - ⇒ **Production way of testing in next page** ⇐ - - - - -

19.1.7. Table Driven Test

⇒ In section '19.1.6. Problems with Normal Testing' we saw the problem with basic testing.
⇒ Now to fix that problem we do:
→ Make a slice with names, parameters and expected values
→ and we run a loop on it and run subtest with those values

```
func TestMath(t *testing.T) {
        cases := []struct {
                name     string
                num1     int
                num2     int
                expected int
        }{
                {name: "Correct Test", num1: 5, num2: 10, expected: 15},
                {name: "Incorrect Test", num1: 2, num2: 2, expected: 7},
        }

        for _, c := range cases {
           t.Run(c.name, func(t *testing.T) {

                result := Add(c.num1, c.num2)      < - - - - - -  Calling the function to test

                if result != c.expected {
                    t.Fatalf("Case %s : expected %d got %d", c.name, c.expected, result)
                }

           })
        }
}
```

- - - - - ⇒ **Now Let's understand how this works** ⇐ - - - - -

Understanding `t.Run()`

⇒ we've already seen that `testing.T` is providing us the method `Run()`

```
type T struct {
        ...
}
func (t *T) Run(name string, f func(t *T)) bool
...
```

⇒ **Parameters:**
→ Name of the test
→ Test function with the parameter (`testing.T`)

⇒ **Returns :**
→ bool

What `t.Run()` does:

⇒ `t.Run()` creates a new `testing.T` instance for the subtest.
⇒ Calls **'provided function'** passing the argument 't *T'.
⇒ Captures the subtest's result (pass/ fail)
⇒ Returns the result as bool
⇒ Reports results aggregated under the parent test

Fails with Subtest (Must Know)

⇒ `FailNow()` or `Fatal()` or `Fatalf()` stops the current test (top-level or subtest) only
⇒ They never stop parent or sibling tests.

Wondering why/how?

Learn more in the next section → '19.1.8. Understanding how `testing` works behind the scene'

Very Important Concept (Do Not Skip)

19.1.8. Understanding how `testing` works behind the scene

⇒ Tests are NOT functions
⇒ Each Tests are execution units bounded to `testing.T`

⇒ A test is defined by:
→ one `*testing.T`
→ one goroutine
→ one execution boundary

Top-Level Test
⇒ So every top-level test (TestXxx) gets it's own `*testing.T`
SubTest
⇒ Every Subtest created with `t.Run()` gets a new, separate `*testing.T` too

Understanding Fail / FailNow / Fatal (Correct Mental Model)

⇒ Failing functions only affect that test that owns that `*testing.T`
⇒ They NEVER affect other tests

⇒ So avoid this Naming Trap:
→ Fatal doesn't mean stops all tests
→ Fatal actually means stop this particular test `*testing.T`

Again

⇒ Each tests gets its own `*testing.T`

19.2. net/http/httptest

⇒ We've learned core of testing in Go but that's not enough for testing our HTTP handers

19.2.1. How we test our HTTP handlers

Direct Explanation

⇒ Testing handlers means simulating requests and capturing their responses.
⇒ You create fake requests and record responses from your handlers to verify correctness.

Layman Explanation

⇒ Handlers need ResponseWriter and *Request to work ... remember?
⇒ They write responses through ResponseWriter ... remember?
⇒ So, we fake both, pass them to handlers, capture the response they sent back, then finally test that output

19.2.2. Three Approaches to test HTTP Handlers

⇒ HTTP handlers in Go can be tested in three distinct ways
→ depending on what you want to verify.

19.2.2.1. In-Memory Handler Testing :

⇒ You just create fake `Request` and `ResponseWriter`
⇒ Then you directly call that Handler with parameter (w, r)
⇒ Used to test single handler logic only

19.2.2.2. Real HTTP Server Testing :

⇒ You spin up a real HTTP server in background (random local port)
⇒ That server contains one or more handlers
⇒ You send a request to it as Client using `http.Client`
→ section `18. Go as Client`
⇒ Slower than in-memory, but Closer to real-world behavior

19.2.2.3. In-Memory Mux testing :

⇒ You just create fake `Request` and `ResponseWriter`
⇒ Same as approach 1
⇒ But instead of calling the handler, you call the mux/router
⇒ Tests routing + middleware + handlers
⇒ No real server, no network

19.2.3. How to create fake `Request`

⇒ `net/http/httptest` package provides us 2 Functions to simulate a **`http.Request`**

```
func NewRequestWithContext(
        ctx context.Context,
        method,
        target string,
        body io.Reader
        ) *http.Request {...}

func NewRequest(
        method,
        target string,
        body io.Reader
        ) *http.Request {...}
```

→Learn deep about ***http.Request** in section '5. http.Request Anatomy'

NewRequestWithContext(ctx, method, target, body) :

⇒ Takes context, method, url and body

⇒ Returns a request instance which acts as a real request

NewRequest(ctx, method, target, body) :

⇒ It's just `NewRequestWithContext()` but with `context.Background()` as ctx

Code Example :

```
data := bytes.NewBufferString(`{"name":"Diwash"}`)

req := httptest.NewRequest("POST", "/users", data)

req.Header.Set("Content-Type", "application/json")
```

⇒ This creates a new Request with:

→ **Method** : Post

→ **URL**: "/users"

→ **Body**: the data we gave

⇒ We've set the Header with Header.Set() too

19.2.4. How to create fake `ResponseWriter`

⇒ We use function named `NewRecoder` provided by `net/http/httptest`

```
func NewRecorder() *ResponseRecorder {...}
```

→ returns *ResponseRecorder
→ *ResponseRecorder satisfies http.ResponseWriter interface

19.2.5. Core Structure of `httptest.responseRecorder`

```
type ResponseRecorder struct {
        Code          int
        HeaderMap     http.Header
        Body          *bytes.Buffer
        Flushed       bool
}

func (rw *ResponseRecorder) Header() http.Header {...}
func (rw *ResponseRecorder) Write(buf []byte) (int, error) {...}
func (rw *ResponseRecorder) WriteString(str string) (int, error) {...}
func (rw *ResponseRecorder) WriteHeader(code int) {...}
func (rw *ResponseRecorder) Flush() {...}
func (rw *ResponseRecorder) Result() *http.Response {...}
```

responseRecorder.Code :
⇒ Accessed in Testing only, not accessed by Handlers
⇒ Status Code is stored here when handlers set code with `WriteHeader()`
⇒ Used for → testing response code

responseRecorder.HeaderMap :
⇒ Accessed in Testing only, not accessed by Handlers
⇒ Headers stays here when handlers set headers with `w.Header()`
⇒ Used for → testing response headers

responseRecorder.Body :
⇒ Accessed in Testing only, not accessed by Handlers
⇒ Buffer that captures the response body when handlers use `Write()`
⇒ Used for → testing response body

responseRecorder.Flushed :
⇒ Accessed in Testing only, not accessed by Handlers
⇒ Indicates whether `Flush()` was called by handler or not

- - - - - - - - Methods - - - - - - - -

Header() http.Header
Write(buf []byte) (int, error)
WriteString(str string)
WriteHeader(code int)
Flush()
⇒ Accessed by Handlers only, not used in Testing

Result() *http.Response :
⇒ Accessed in Testing only, not accessed by Handlers
⇒ Constructs and returns an `http.Response` object representing the recorded respons
⇒ used for advanced inspection

- - - - - - ⇒ **We'll see the example of using it in coming sections** ⇐ - - - - - -

19.2.6. How To Create MINI Server for Approach 2

⇒ httptest provides us a function `NewServer`
⇒ `httptest.NewServer()` runs a real HTTP server on a random local port
→ Used only inside test

Core Structure

```
type Server struct {
        URL     string          < - - - - - - - - - Base URL, for eg: "127.0.0.1:8080"
        TLS *tls.Config         < - - - - - - - - - nil in non-TLS server
        Config *http.Server     < - - - - - - - - - URL
        ...
}

func NewServer(handler http.Handler) *Server {...}
func NewTLSServer(handler http.Handler) *Server {...}
func (s *Server) Close() {...}
```

type Server struct { ... } :
⇒ This represents the mini HTTP server running in your test
⇒ It holds info about where it's running and lets you control it.

Server.URL :
⇒ The address where the server is listening
⇒ Includes protocol, host, and port
⇒ Example: "http://127.0.0.1:1234"

NewServer() :
⇒ Starts our mini HTTP server with the handler/mux you give
⇒ Starts server as background goroutine

NewTLSServer() :
⇒ Same as `NewServer()` but HTTPS server

server.Close() :
⇒ Stops the server and frees the port
⇒ !!! DO NOT Forget this !!!

Example Code

```
                    * - - - - - - Creating our Mini Server  - - - - - - *
server := httptest.NewServer(YourHandler)
defer server.Close()

                    * - - - - - - Sending Client request to Mini Server - - - - - - *
resp, err := http.Get(server.URL)
if err != nil {...}
defer resp.Body.Close()

                    * - - - - - - Testing the Response - - - - - - *
if resp.StatusCode != http.StatusOK {
   t.Fatalf("expected 200, got %d", resp.StatusCode)
}
```

In Layman:

1st ⇒ We just ran new mini server with the handler we're about to test

2nd ⇒ We sent a client request to that server and captured the response

3rd ⇒ Finally, tested the response

19.2.7. [Example]: Approach 1 - In-Memory Handler Testing

The function we will be testing :

```
func HandleUser(w http.ResponseWriter, r *http.Request) {
        defer r.Body.Close()

        var user User
        json.NewDecoder(r.Body).Decode(&user)

        if len(user.Password) < 8 || len(user.Username) < 3 {
                w.WriteHeader(http.StatusBadRequest)
                return
        }
        w.WriteHeader(http.StatusCreated)
}
```

Testing Example :

```
func TestUserHandler(t *testing.T) {
   data := []byte(`{"username":"Diwash","password":"password"}`)   <- - - -  json string

   var req *http.Request = httptest.NewRequest(http.MethodPost, "/",
bytes.NewBuffer(data))
   req.Header.Set("Content-Type", "application/json")
   var w *httptest.ResponseRecorder = httptest.NewRecorder()

   HandleUser(w, req)

   if w.Code != http.StatusCreated {
        t.Errorf("Expected %d, Got %d", http.StatusCreated, w.Code)
   }
}
```

What's happening in the code ?

⇒ Created fake json string request body
⇒ Created a fake HTTP Post Request
⇒ Created fake `ResponseWriter` for handler with `NewRecorder()`
⇒ Called handler func with fake requests and writer
⇒ Then Tested the result accessing the `Code` from `*responseRecorder`

19.2.8. [Example]: Approach 2 - Mini HTTP Server

The function we will be testing :

```
func HelloHandler(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(http.StatusOK)
        w.Write([]byte("Hello World"))
}
```

Testing Example :

```
                    * - - - - - - Creating our Mini Server  - - - - - - *
server := httptest.NewServer(HelloHandler)
defer server.Close()

                    * - - - - - - Sending Client request to Mini Server - - - - - - *
resp, err := http.Get(server.URL)
if err != nil {...}
defer resp.Body.Close()

                    * - - - - - - Testing the Response - - - - - - *
if resp.StatusCode != http.StatusOK {
   t.Fatalf("expected 200, got %d", resp.StatusCode)
}
```

19.2.9. [Example]: Approach 3 - In-Memory Mux testing

⇒ Yup we can directly test the mux too, since we know it's just like a Handler.

→ check section '2.3.6. [IMPORTANT] : Deep Dive into http.ServeMux and http.Server'

The mux we will be testing:

```
var Mux *http.ServeMux

func main(){
        Mux = http.NewServeMux()
        Mux.HandleFunc("/health", HealthHandler)
        Mux.HandleFunc("/user", UserHandler)
        ...
}
func HealthHandler(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(http.StatusOK)
        w.Write([]byte("OK"))
}
func UserHandler(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(http.StatusCreated)
        w.Write([]byte("User Created"))
}
```

Testing Example :

```
func TestMuxHandlers(t *testing.T) {

        req := httptest.NewRequest("GET", "/health", nil)
        rr := httptest.NewRecorder()
        Mux.ServeHTTP(rr, req)      <- - - -  Main mux of main server

        if rr.Code != http.StatusOK {
                t.Errorf("Expected %d, Got %d", http.StatusOK, rr.Code)
        }
}
```

What's Happening?

⇒ We created our server as usual, right?

⇒ We created fake `Request` and `ResponseWriter`

⇒ And passed it directly to the `Mux`

→ Remember? Mux is a Handler too

→ check section '4.6. [IMPORTANT] : Deep Dive into http.ServeMux and http.Server'

That's it for now, my friend. But Wait

If you made it here... respect. Genuinely.

You thought this was a book about Go backend, didn't you?

It is. But it isn't.

Think about what actually happened here. Every single time I introduced something new, I didn't start with the code. I started with why it exists. TCP before HTTP. Bytes before structs. The problem before the solution. Every time.

That wasn't about Go.

That was me quietly teaching you how to learn anything.

You just spent this entire book practicing first principles thinking without realizing it. Every chapter was a rep. And now it's in your muscle memory. You'll never look at a new library, a new language, a new framework the same way again. You'll automatically ask ‘okay but how does this actually work underneath?’.

That question is worth more than everything I taught you about Go.

Go will change. Packages will update. New frameworks will come. But that question... that habit of going deeper, that stays with you forever.

AND, don’t just apply it in Go. Apply it in every area of your life, my friend. Break everything to its core. And understand how things actually are.

So yeah. You learned Go backend. But that was never really the point.

— Diwash Pandey

If this helped you, you can support for more of this here

buymemomo.com/diwashpandey

Thank You

Credits & Attribution

“Go” is a trademark of Google LLC.

The Go gopher was originally created by Renée French and is licensed under the Creative Commons Attribution 4.0 License:
https://creativecommons.org/licenses/by/4.0/

Go project branding and trademark information:
https://go.dev/brand

This book is an independent educational work and is not affiliated with or endorsed by Google or the Go project.