Gorgo Go for Java Programmers

June 11, 2026

Contents

Chapter 15

JSON, HTTP, and the Web

Go's standard library ships everything you need to build and consume web services — no Spring Boot, no Jackson, no Tomcat required. In the Java world, a production HTTP service usually drags in a web framework, a servlet container, and a JSON binding library, each with its own configuration files and version matrix. In Go, `encoding/json` and `net/http` cover the same ground from the standard library: a routed HTTP server with method matching, middleware, TLS, and JSON binding fits in a single file with no external dependencies. That keeps your `go.mod` short, your builds fast, and your deployment a single static binary. This chapter covers JSON encoding and decoding, making HTTP requests, building HTTP servers with the 1.22 mux, writing middleware, live profiling endpoints with `net/http/pprof`, XML encoding, raw TCP/UDP networking with the `net` package, and securing connections with `crypto/tls`.

Encoding JSON

Java programmers typically reach for Jackson or Gson to serialize objects to JSON. In Go, `encoding/json` is in the standard library and needs no configuration beyond struct tags.

The two core functions are:

```
func Marshal(v any) ([]byte, error)       // encode v as JSON bytes
func Unmarshal(data []byte, v any) error  // decode JSON bytes into v (non-nil pointer)
```

Here is a round trip:

```
package main

import (
    "encoding/json"
    "fmt"
)

type Song struct {
    Title  string `json:"title"`
    Artist string `json:"artist"`
    BPM    int    `json:"bpm"`
}

func main() {
    s := Song{Title: "Sandstorm", Artist: "Darude", BPM: 136}

    data, err := json.Marshal(s)
```

```
    if err != nil {
        panic(err)
    }
    fmt.Println(string(data))
    // {"title":"Sandstorm","artist":"Darude","bpm":136}

    var s2 Song
    if err := json.Unmarshal(data, &s2); err != nil {
        panic(err)
    }
    fmt.Printf("%+v\n", s2)
    // {Title:Sandstorm Artist:Darude BPM:136}
}
```

`Marshal` returns a `[]byte`. Convert it to `string` with `string(data)` when you need to print it or embed it in a larger string. `Unmarshal` takes a pointer so it can write into the struct; passing a non-pointer returns an error.

Tip: In Jackson you annotate fields with `@JsonProperty("title")`. In Go you use a struct tag: `` `json:"title"` ``. Both achieve the same mapping — Go's approach requires no annotation processor or reflection framework beyond what `encoding/json` already does at runtime.

Struct Tags for JSON

A struct tag is a raw string literal attached to a field declaration. `encoding/json` reads the `json` key in each tag to control marshaling.

```
type Track struct {
    Title    string  `json:"title"`                // rename to "title"
    Artist   string  `json:"artist"`               // rename to "artist"
    BPM      int     `json:"bpm,omitempty"`        // omit if BPM == 0
    Internal string  `json:"-"`                    // always skip
}
```

The options after the comma are:

- `omitempty` — skip the field when its value is empty: `false`, `0`, a `nil` pointer or interface, or any empty array, slice, map, or string.
- `-` — never include this field in JSON output, even if it has a value.

```
t1 := Track{Title: "Out Of The Blue", Artist: "System F", BPM: 138, Internal: "secret"}
data, _ := json.Marshal(t1)
fmt.Println(string(data))
// {"title":"Out Of The Blue","artist":"System F","bpm":138}
// Internal is gone; BPM present because it is non-zero

t2 := Track{Title: "Flaming June", Artist: "BT"}
data, _ = json.Marshal(t2)
fmt.Println(string(data))
// {"title":"Flaming June","artist":"BT"}
// BPM omitted because it is zero (omitempty)
```

Trap: `omitempty` has no effect on a struct-typed field — a zero-valued struct is never omitted. If you want a nested struct to be omittable, make the field a pointer: `` `json:"meta,omitempty"` `` on a `*Meta` field will omit `meta` when the pointer is `nil`. Since Go 1.24 there is a cleaner fix: the `omitzero` option (`` `json:"meta,omitzero"` ``) omits any field whose value is the zero value for its type — including zero-valued structs — no pointer required. If the type has an `IsZero() bool` method, `omitzero` uses it, which is how a zero `time.Time` gets omitted.

Tag syntax is strict: backtick string, `key:"value"` pairs separated by spaces, no commas between pairs. `go vet ./...` catches malformed tags.

Streaming with `Encoder` and `Decoder`

`Marshal` and `Unmarshal` work on in-memory byte slices. When the JSON is coming from or going to an `io.Reader` or `io.Writer` — an HTTP body, a file, a network connection — use `json.NewEncoder` and `json.NewDecoder` instead.

```
func NewEncoder(w io.Writer) *Encoder  // write JSON to w
func NewDecoder(r io.Reader) *Decoder  // read JSON from r
```

Key methods on those types:

```
func (enc *Encoder) Encode(v any) error      // marshal v and write to w, followed by a newline
func (dec *Decoder) Decode(v any) error      // read one JSON value from r and unmarshal into v
func (dec *Decoder) More() bool              // true if another value remains in the stream
```

Encoding directly to an `http.ResponseWriter`:

```
func songHandler(w http.ResponseWriter, r *http.Request) {
    s := Song{Title: "Saltwater", Artist: "Chicane", BPM: 138}
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(s)
}
```

Decoding a stream of newline-delimited JSON records:

```
dec := json.NewDecoder(r.Body)
for dec.More() {
    var s Song
    if err := dec.Decode(&s); err != nil {
        break
    }
    fmt.Println(s.Title)
}
```

By default the decoder silently ignores JSON fields that have no matching struct field. To reject them instead, call `DisallowUnknownFields` before decoding:

```
func (dec *Decoder) DisallowUnknownFields()  // make Decode fail on unknown JSON fields
```

```
dec := json.NewDecoder(r.Body)
dec.DisallowUnknownFields()  // any extra field makes Decode return an error
```

This is the analog of Jackson's `FAIL_ON_UNKNOWN_PROPERTIES` for Java readers.

Tip: Prefer streaming `Encoder`/`Decoder` over `Marshal`/`Unmarshal` when working with `io.Reader`/`io.Writer`. Streaming avoids allocating the entire JSON payload as a byte slice, which matters for large payloads.

Making HTTP Requests

The `net/http` package is also the HTTP client. The simplest way to make a GET request is `http.Get`:

```
func Get(url string) (resp *Response, err error)  // GET url; caller must close resp.Body
```

```
resp, err := http.Get("https://api.example.com/songs/1")
if err != nil {
    log.Fatal(err)
}
defer resp.Body.Close()  // MUST close the body

body, err := io.ReadAll(resp.Body)
if err != nil {
    log.Fatal(err)
}
fmt.Println(string(body))
```

Trap: You **must** close `resp.Body`, even if you do not read it. Failing to close the body leaks the underlying TCP connection and eventually exhausts the connection pool. `defer resp.Body.Close()` immediately after checking `err` is the standard idiom.

For anything beyond a simple GET — custom headers, timeouts, POST bodies — use `http.Client` and `http.NewRequest`:

```
// http.Client --- reusable, configurable; zero value uses sensible defaults
type Client struct {
    Transport     RoundTripper  // how to make a single HTTP transaction
    CheckRedirect func(req *Request, via []*Request) error  // redirect policy
    Jar           CookieJar     // cookie storage
    Timeout       time.Duration // zero means no timeout
}

func NewRequest(method, url string, body io.Reader) (*Request, error)  // build a request
func (c *Client) Do(req *Request) (*Response, error)                    // execute the request
```

A POST with a JSON body and a timeout:

```
client := &http.Client{Timeout: 10 * time.Second}

song := Song{Title: "Out Of The Blue", Artist: "System F", BPM: 138}
buf := new(bytes.Buffer)
json.NewEncoder(buf).Encode(song)

req, err := http.NewRequest("POST", "https://api.example.com/songs/", buf)
if err != nil {
    log.Fatal(err)
}
req.Header.Set("Content-Type", "application/json")

resp, err := client.Do(req)
if err != nil {
    log.Fatal(err)
}
defer resp.Body.Close()
```

```
fmt.Println(resp.StatusCode)  // e.g. 201
```

Tip: Create one `http.Client` and reuse it across requests. `http.Client` manages a connection pool (`Transport`) internally. Creating a new client per request bypasses pooling and creates a new pool each time, which is wasteful and slower. The package-level `http.Get` and `http.Post` helpers use a shared default client (`http.DefaultClient`).

In Java you would use `HttpClient` (Java 11+), `RestTemplate`, or OkHttp. Go's `http.Client` fills the same role.

Building an HTTP Server

Go has a built-in HTTP server that calls out to an `http.Handler` to service the requests. The simplest way to start up an HTTP server is `http.ListenAndServe`, which takes a `host:port` to listen on and an `http.Handler` to service the requests.

```
func ListenAndServe(addr string, handler Handler) error  // nil handler = DefaultServeMux
```

`http.ListenAndServe` is really just for quick demos and prototypes. It spins up a server with all the default settings and gives you no handle to control it once it is running — there is no way to set timeouts, no way to shut it down gracefully, no way to do anything but kill the process. For production setups you instantiate an `http.Server` yourself, which gives you a handle to control the server's lifecycle (configure timeouts, shut down gracefully, drain in-flight requests). We start with `http.ListenAndServe` because it keeps the early examples short, then switch to `http.Server` once we have something worth running for real.

The foundation of Go's HTTP server is the `http.Handler` interface:

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)  // handles one HTTP request
}
```

Here is a super basic HTTP server:

```
package main

import (
    "fmt"
    "net/http"
)

func serveHttpRequest(w http.ResponseWriter, r *http.Request) {
    _, _ = w.Write([]byte("Hello from " + r.URL.Path))
}
func main() {
    err := http.ListenAndServe(":8888", http.HandlerFunc(serveHttpRequest))
    if err != nil {
        fmt.Printf("Problem starting http server %v", err)
    }
}
```

Before we jump into the HTTP API, it is worthwhile to take a moment to highlight how useful adapter functions are. `http.HandlerFunc` is a function type that has the same signature as `serveHttpRequest`. It also has a method, `func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request)`, so `http.HandlerFunc` implements the `http.Handler` interface, which is why we can pass it to `http.ListenAndServe`.

For nontrivial HTTP apps, we need better server management and request routing. In this simple example, we look at the path of the request in the handler function. Ideally, we would like to route requests to different

handler functions based on the type of request and the request path. Go has us covered! A **mux** (short for *multiplexer*, also called a *router*) is itself a handler whose whole job is to look at the request's method and path and dispatch to the right handler. `http.ServeMux` is the standard library's implementation. Think of it as `web.xml`/`@RequestMapping` routing, but as plain code you write in `main`.

When we use a mux, the request flow is: the network connection lands in the server, the server hands the request to the mux, and the mux dispatches it to the handler registered for that route. Because every layer is just an `http.Handler`, you can wrap one in another — which is exactly how middleware works, as you'll see later in this chapter.

In the `ServeMux` example below we will also see the `http.Server` type, which lets us construct a server that can be configured, started, and stopped.

`ServeMux` — Method Routing and Wildcards

`http.ServeMux` is a builtin handler that routes requests to `http.Handler`s or plain handler functions based on the request method and path, and can even parse out path elements. Before Go 1.22, `http.ServeMux` matched only on path prefixes and required external routers (Chi, Gorilla Mux, etc.) for method or wildcard routing. Go 1.22 upgraded the built-in mux significantly.

Method routing — prefix the pattern with an HTTP method and a space:

```
mux := http.NewServeMux()
mux.HandleFunc("GET /songs/", listSongs)    // only GET requests
mux.HandleFunc("POST /songs/", createSong)  // only POST requests
```

Path wildcards — `{name}` captures a path segment, `{name...}` captures the rest:

```
mux.HandleFunc("GET /songs/{id}/", getSong)  // /songs/42/ --- id = "42"
```

Retrieve the captured value with `r.PathValue`:

```
func getSong(w http.ResponseWriter, r *http.Request) {
    id := r.PathValue("id")                        // "42"
    fmt.Fprintf(w, "fetching song %s\n", id)
}
```

A complete example wiring up a small songs API:

```
package main

import (
    "encoding/json"
    "log"
    "net/http"
    "time"
)

type Song struct {
    ID     string `json:"id"`
    Title  string `json:"title"`
    Artist string `json:"artist"`
}

var catalog = map[string]Song{
    "1": {ID: "1", Title: "Sandstorm", Artist: "Darude"},
    "2": {ID: "2", Title: "Saltwater", Artist: "Chicane"},
}
```

```go
func getSong(w http.ResponseWriter, r *http.Request) {
    id := r.PathValue("id")
    song, ok := catalog[id]
    if !ok {
        http.Error(w, "not found", http.StatusNotFound)
        return
    }
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(song)
}

func listSongs(w http.ResponseWriter, r *http.Request) {
    songs := make([]Song, 0, len(catalog))
    for _, s := range catalog {
        songs = append(songs, s)
    }
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(songs)
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /songs/",      listSongs)
    mux.HandleFunc("GET /songs/{id}/", getSong)

    srv := &http.Server{
        Addr:         ":8080",
        Handler:      mux,
        ReadTimeout:  5 * time.Second,   // max time to read the whole request
        WriteTimeout: 10 * time.Second,  // max time to write the response
        IdleTimeout:  60 * time.Second,  // max keep-alive idle time
    }
    log.Fatal(srv.ListenAndServe())     // srv is the handle to the running server
}
```

Instead of calling the package-level `http.ListenAndServe`, we build an `http.Server` value and call its `ListenAndServe` method. The `srv` variable is now a handle to the running server: it carries the timeouts we configured and, as you will see in a moment, lets us shut the server down on our own terms.

```go
func (srv *Server) ListenAndServe() error              // listen on srv.Addr and serve
func (srv *Server) Shutdown(ctx context.Context) error  // graceful stop, drains in-flight
func (srv *Server) Close() error                        // immediately close all connections
```

Trap: The package-level `http.ListenAndServe` has no timeouts at all. A slow or malicious client that opens a connection and never finishes its request ties up a goroutine indefinitely (a *Slowloris* attack). Always set `ReadTimeout` and `WriteTimeout` on a real `http.Server`.

To stop cleanly, run the server in a goroutine and call `Shutdown` when you catch a termination signal. `Shutdown` stops accepting new connections and waits for in-flight requests to finish, up to the context's deadline:

```go
func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /songs/",      listSongs)
    mux.HandleFunc("GET /songs/{id}/", getSong)
```

```
    srv := &http.Server{Addr: ":8080", Handler: mux}

    go func() {
        if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
            log.Fatalf("server error: %v", err)
        }
    }()

    // wait for SIGINT or SIGTERM
    stop := make(chan os.Signal, 1)
    signal.Notify(stop, os.Interrupt, syscall.SIGTERM)
    <-stop

    // give in-flight requests up to 10 seconds to finish
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
    defer cancel()
    if err := srv.Shutdown(ctx); err != nil {
        log.Printf("graceful shutdown failed: %v", err)
    }
}
```

`ListenAndServe` returns `http.ErrServerClosed` once `Shutdown` is called, which is why we treat that error as the normal exit rather than a failure. This is the Go equivalent of a servlet container draining requests before stopping — but it is a few lines of explicit code rather than container lifecycle hooks.

Notice that `getSong` calls `http.Error` and immediately returns when the song is not found, so the success path only runs when the lookup succeeds. [*error-first-return-early*]

When both a wildcard pattern (`GET /songs/{id}/`) and a subtree pattern (`GET /songs/`) could match a request, the more-specific pattern wins, so `/songs/42/` goes to `getSong` rather than `listSongs`.

Wut: A trailing slash in the pattern makes it a subtree match: `GET /songs/` matches `/songs/`, `/songs/anything`, and `/songs/42/`. Without the trailing slash, `GET /songs` matches only the exact path `/songs`. This is the same behavior as pre-1.22 `ServeMux`.

Trap: A `{id}/` (trailing-slash) pattern requires the trailing slash, so a request to `/songs/1` is redirected (a 307 to `/songs/1/`), which can surprise clients that do not follow redirects.

Compared to Java: this is the rough equivalent of Spring MVC's `@GetMapping("/songs/{id}")` or JAX-RS's `@GET @Path("/songs/{id}")`, but built into the standard library with no framework dependency.

Middleware Chaining

Middleware in Go is a function that takes an `http.Handler` and returns an `http.Handler`. It wraps the inner handler with cross-cutting logic — logging, authentication, metrics, CORS — without modifying the handler itself.

The pattern:

```
func logging(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        log.Printf("%s %s", r.Method, r.URL.Path)  // before
        next.ServeHTTP(w, r)                       // call the inner handler
    })
}
```

Stack middleware by composing the wrappers:

```
func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /songs/{id}/", getSong)

    handler := logging(mux)   // outermost middleware runs first
    http.ListenAndServe(":8080", handler)
}
```

Add more layers by nesting calls:

```
handler := logging(auth(mux))  // logging wraps auth wraps mux
```

Tip: The middleware pattern is the Go equivalent of Java's servlet filters or Spring's `HandlerInterceptor`. Because `http.Handler` is a simple interface with one method, any function that wraps it composes cleanly — no framework annotation or XML configuration required.

You can also write a `chain` helper to apply a list of middleware in order:

```
func chain(h http.Handler, middleware ...func(http.Handler) http.Handler) http.Handler {
    for i := len(middleware) - 1; i >= 0; i-- {
        h = middleware[i](h)       // apply in reverse so the first listed runs first
    }
    return h
}

handler := chain(mux, logging, auth, metrics)
```

Live Profiling with `net/http/pprof`

Go's runtime can expose profiling data — CPU usage, heap allocations, goroutine stacks, mutex contention — as HTTP endpoints. All it takes is a blank import.

```
import _ "net/http/pprof"  // register pprof handlers with DefaultServeMux
```

The side-effect import registers several routes under `/debug/pprof/` on `http.DefaultServeMux`. If your server already calls `http.ListenAndServe(addr, nil)` (which uses `DefaultServeMux`), those routes are live the moment the process starts — no code changes beyond the import.

If you are using a custom `*http.ServeMux`, register the handlers explicitly:

```
import (
    "net/http"
    "net/http/pprof"
)

mux := http.NewServeMux()
mux.HandleFunc("GET /songs/{id}/", getSong)

// register pprof endpoints on the same mux
mux.HandleFunc("GET /debug/pprof/",        pprof.Index)
mux.HandleFunc("GET /debug/pprof/cmdline", pprof.Cmdline)
mux.HandleFunc("GET /debug/pprof/profile", pprof.Profile)
mux.HandleFunc("GET /debug/pprof/symbol",  pprof.Symbol)
mux.HandleFunc("GET /debug/pprof/trace",   pprof.Trace)
```

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

The key endpoints under `/debug/pprof/`:

Endpoint	What it shows
`/`	index of available profiles
`heap`	live heap allocations
`goroutine`	stack trace of every live goroutine
`allocs`	all past allocations, including freed ones
`mutex`	stack traces of code that held contended mutexes
`block`	stack traces of goroutines blocked on synchronization primitives (channel ops, mutex contention, `sync.WaitGroup`, etc.)
`profile?seconds=N`	CPU profile for N seconds (default 30)

The `block` and `mutex` profiles are off by default — call `runtime.SetBlockProfileRate(1)` and `runtime.SetMutexProfileFraction(1)` (typically at startup, behind a debug flag) to enable them.

Once the server is running, collect and view a profile with `go tool pprof`:

```
# collect a 30-second CPU profile and open the interactive browser UI
go tool pprof -http=:6060 http://localhost:8080/debug/pprof/profile?seconds=30

# inspect the live heap
go tool pprof -http=:6060 http://localhost:8080/debug/pprof/heap
```

`go tool pprof -http` opens a flame graph and call-graph view in your browser. The flame graph makes it easy to see which functions consume the most CPU or allocate the most memory.

Trap: Never expose `/debug/pprof/` on a public-facing port. CPU profiling adds measurable overhead to the whole process while it runs, the endpoints accept unbounded concurrent requests, and the profiles leak internals — source paths, goroutine stacks, even heap contents — to anyone who can reach the port. The standard pattern is to run pprof on a separate, internal-only port:

```
// internal admin server on a non-public port
go func() {
    // nil handler means DefaultServeMux, which has the pprof routes
    log.Println(http.ListenAndServe("localhost:6060", nil))
}()
// public server with a custom mux
http.ListenAndServe(":8080", mux)
```

In a container environment, expose port 6060 only within the cluster network.

Tip: In Java, runtime profiling typically requires attaching an external tool (VisualVM, JProfiler, YourKit) via the JVM attach mechanism or JMX. `net/http/pprof` is the Go equivalent built into the process — no agent, no attach, no separate daemon. The profile data is available over plain HTTP, which means you can script it with `curl` or integrate it into any monitoring pipeline.

Encoding XML

`encoding/xml` works exactly like `encoding/json` — same `Marshal`/`Unmarshal` functions, same streaming `Encoder`/`Decoder`, same struct tag mechanism. The tag key is `xml` instead of `json`.

```
func Marshal(v any) ([]byte, error)              // encode v as XML
func Unmarshal(data []byte, v any) error         // decode XML into v
func NewEncoder(w io.Writer) *Encoder            // streaming XML encoder
func NewDecoder(r io.Reader) *Decoder            // streaming XML decoder
```

```
import "encoding/xml"

type Song struct {
    XMLName xml.Name `xml:"song"`           // set the element name
    Title   string   `xml:"title"`
    Artist  string   `xml:"artist"`
    BPM     int      `xml:"bpm,omitempty"`
}

s := Song{Title: "Flaming June", Artist: "BT", BPM: 138}
data, _ := xml.Marshal(s)
fmt.Println(string(data))
// <song><title>Flaming June</title><artist>BT</artist><bpm>138</bpm></song>
```

The `xml.Name` field with tag `` `xml:"song"` `` sets the root element name. Without it, `encoding/xml` uses the struct type name, which is often not what you want.

XML tag options beyond the field name:

- `xml:"name,attr"` — encode the field as an XML attribute rather than a child element.
- `xml:",chardata"` — encode the field as the character data content of the parent element.
- `xml:",omitempty"` — same as JSON: omit when zero value.
- `xml:"-"` — always skip.

Tip: If your service needs both JSON and XML representations of the same struct, define both tag keys on each field:

```
type Song struct {
    Title  string `json:"title"  xml:"title"`
    Artist string `json:"artist" xml:"artist"`
}
```

Both `encoding/json` and `encoding/xml` read only their own tag key and ignore the other.

Raw Networking with net

HTTP sits on top of TCP. The `net` package gives you direct access to TCP and UDP connections — useful for non-HTTP protocols, custom servers, or low-level networking tools.

The two fundamental functions are:

```
func Dial(network, address string) (Conn, error)   // connect to a server
func Listen(network, address string) (Listener, error) // listen for incoming connections
```

`network` is `"tcp"`, `"tcp4"`, `"tcp6"`, `"udp"`, `"udp4"`, or `"udp6"` for `Dial`; `Listen` accepts only the stream-oriented networks (`"tcp"` variants and `"unix"`). `address` is `"host:port"` for TCP/UDP, e.g. `"localhost:8080"` or `":8080"` (all interfaces).

`net.Conn` implements both `io.Reader` and `io.Writer`, so you can use any `io` utility on it:

```
// net.Conn is an interface; these are its method signatures
type Conn interface {
    Read(b []byte) (n int, err error)        // receive bytes
    Write(b []byte) (n int, err error)       // send bytes
```

```
    Close() error                          // close the connection
    LocalAddr() Addr                       // this side's network address
    RemoteAddr() Addr                      // the peer's network address
    SetDeadline(t time.Time) error         // set read+write deadline
    SetReadDeadline(t time.Time) error     // set read deadline only
    SetWriteDeadline(t time.Time) error    // set write deadline only
}
```

`LocalAddr` and `RemoteAddr` return a `net.Addr`, a small interface with `Network() string` (the protocol, e.g. `"tcp"`) and `String() string` (the `host:port` text); `conn.RemoteAddr().String()` is the usual way to log who connected.

A simple TCP echo client:

```
package main

import (
    "fmt"
    "net"
)

func main() {
    conn, err := net.Dial("tcp", "localhost:9000")
    if err != nil {
        panic(err)
    }
    defer conn.Close()

    fmt.Fprintf(conn, "hola\n")     // send

    buf := make([]byte, 1024)
    n, _ := conn.Read(buf)          // receive
    fmt.Printf("echo: %s", buf[:n])
}
```

Tip: Plain `net.Dial` has no connection-setup timeout — if the host is unreachable the call can block for the operating system's default, often a minute or more. The connection deadlines below (`SetReadDeadline` and friends) do not help here, because they only apply *after* the connection exists. To bound the dial itself, reach for `net.DialTimeout`:
`func DialTimeout(network, address string, timeout time.Duration) (Conn, error)`
`conn, err := net.DialTimeout("tcp", "localhost:9000", 5*time.Second)`
When you need `context`-based cancellation instead of a fixed duration, use a `net.Dialer` and its `DialContext` method: `(&net.Dialer{}).DialContext(ctx, "tcp", addr)`.

A minimal TCP echo server:

```
package main

import (
    "io"
    "net"
)

func main() {
    ln, err := net.Listen("tcp", ":9000")
    if err != nil {
```

```
        panic(err)
    }
    defer ln.Close()

    for {
        conn, err := ln.Accept()    // block until a client connects
        if err != nil {
            break
        }
        go io.Copy(conn, conn)      // echo: copy reads back to writes
    }
}
```

`ln.Accept()` blocks until a new connection arrives and returns a `net.Conn` for that connection. Each connection is handled in its own goroutine so the server can accept the next connection immediately.

Wut: `io.Copy(conn, conn)` works because `net.Conn` implements both `io.Reader` and `io.Writer`. `io.Copy(dst, src)` reads from `src` and writes to `dst` until EOF or an error. Here `conn` is both — every byte received is immediately written back.

Deadlines

`conn.Read` and `conn.Write` block indefinitely by default — a silent peer can hang a goroutine forever. In Java, you bound each read with `socket.setSoTimeout(5000)`. After that call, any read that blocks for more than 5000 milliseconds returns a timeout. Go takes a different approach: instead of a *duration* you set an absolute *deadline* with `SetDeadline`, `SetReadDeadline`, or `SetWriteDeadline`, each taking a `time.Time`. Once that deadline has passed, all reads on that connection will fail. So instead of setting a 5 second timeout, we say 5 seconds from now any reads on the connection will fail:

```
conn.SetReadDeadline(time.Now().Add(5 * time.Second))   // fail if no data within 5s
n, err := conn.Read(buf)
```

When the deadline passes, the blocked `Read` or `Write` returns immediately with an error that satisfies `net.Error` and reports `Timeout() == true` (it wraps `os.ErrDeadlineExceeded`):

```
n, err := conn.Read(buf)
if err != nil {
    var ne net.Error
    if errors.As(err, &ne) && ne.Timeout() {
        // deadline hit --- handle the timeout
    }
}
```

Wut: Deadlines are *absolute*, not rolling. `SetReadDeadline` sets one fixed instant, not a per-read timer like Java's `setSoTimeout`. For an idle timeout that resets on every successful read, you have to call `SetReadDeadline(time.Now().Add(d))` again before *each* read.

A deadline applies to all future I/O on the connection, not just the next call, until you change it. Pass the zero value, `time.Time{}`, to clear a deadline and block indefinitely again:

```
conn.SetDeadline(time.Time{})   // no deadline
```

Tip: Unlike most blocking operations in Go, `net.Conn` I/O is *not* cancelled by a `context.Context`. If you have a `ctx` with a deadline, bridge it to the connection with `conn.SetDeadline(deadline)` where `deadline, ok := ctx.Deadline()`.

For UDP, dial with `net.Dial("udp", ...)` as before, but `net.Listen` does not accept UDP — it only handles stream-oriented networks. On the server side use `net.ListenPacket("udp", ...)`, which returns a `net.PacketConn` that tracks each client by address.

Tip: In Java, raw TCP/UDP programming uses `java.net.Socket`, `ServerSocket`, `DatagramSocket`, and `DatagramPacket`. Go's `net.Conn` is a simpler unified abstraction: it is an `io.Reader` and `io.Writer`, so you can layer a `bufio.Scanner` or a `json.Decoder` directly on top of it without any adapter code.

TLS with `crypto/tls`

The `crypto/tls` package provides TLS 1.2 and 1.3 support for both servers and clients. For HTTP, the standard library handles TLS transparently — you mostly just point it at your certificate files. For raw TCP, `crypto/tls` wraps a `net.Conn` into an encrypted connection that still satisfies `io.Reader` and `io.Writer`.

HTTPS Server

The simplest way to add TLS to an HTTP server is `http.ListenAndServeTLS`:

```
func ListenAndServeTLS(addr, certFile, keyFile string, handler Handler) error
```

```
mux := http.NewServeMux()
mux.HandleFunc("GET /songs/", listSongs)
http.ListenAndServeTLS(":443", "cert.pem", "key.pem", mux)
```

For more control, build a `tls.Config` and pass it to `http.Server`:

```
cert, err := tls.LoadX509KeyPair("cert.pem", "key.pem")  // load certificate and private key
if err != nil {
    log.Fatal(err)
}
srv := &http.Server{
    Addr:    ":443",
    Handler: mux,
    TLSConfig: &tls.Config{
        Certificates: []tls.Certificate{cert},  // one or more certificates
        MinVersion:   tls.VersionTLS12,          // reject TLS 1.0 and 1.1
    },
}
srv.ListenAndServeTLS("", "")  // cert and key already in TLSConfig; pass empty strings
```

TLS Client Configuration

The default `http.Client` already validates server certificates against the system trust store — you get HTTPS for free just by using an `https://` URL. To customize TLS behavior, set `TLSClientConfig` on `http.Transport`:

```
client := &http.Client{
    Transport: &http.Transport{
        TLSClientConfig: &tls.Config{
            MinVersion: tls.VersionTLS12,  // require TLS 1.2+
        },
    },
}
```

To trust a custom certificate authority (common in internal services):

```
pool := x509.NewCertPool()
caCert, _ := os.ReadFile("internal-ca.pem")
pool.AppendCertsFromPEM(caCert)

client := &http.Client{
    Transport: &http.Transport{
        TLSClientConfig: &tls.Config{
            RootCAs: pool,  // trust this CA in addition to the system store
        },
    },
}
```

Trap: `tls.Config{InsecureSkipVerify: true}` disables all certificate validation. It is occasionally used in local development against self-signed certs, but it silently makes man-in-the-middle attacks undetectable. Never use it in production code, and never commit it to a shared repository.

Raw TLS Connections

`crypto/tls` mirrors the `net` package: `tls.Dial` and `tls.Listen` replace `net.Dial` and `net.Listen` and return a `*tls.Conn`, which implements `net.Conn` — and therefore `io.Reader` and `io.Writer`.

```
func Dial(network, addr string, config *tls.Config) (*tls.Conn, error)        // TLS client
func Listen(network, laddr string, config *tls.Config) (net.Listener, error)  // TLS listener
```

A TLS client connecting to a server:

```
conn, err := tls.Dial("tcp", "api.example.com:443", &tls.Config{
    MinVersion: tls.VersionTLS12,
})
if err != nil {
    log.Fatal(err)
}
defer conn.Close()

fmt.Fprintf(conn, "GET / HTTP/1.0\r\nHost: api.example.com\r\n\r\n")
io.Copy(os.Stdout, conn)
```

`tls.NewListener` wraps an existing `net.Listener` to add TLS to any protocol:

```
func NewListener(inner net.Listener, config *tls.Config) net.Listener

ln, _ := net.Listen("tcp", ":9443")
tlsLn := tls.NewListener(ln, &tls.Config{Certificates: []tls.Certificate{cert}})
for {
    conn, _ := tlsLn.Accept()   // conn is a *tls.Conn wrapping a net.Conn
    go handleConn(conn)
}
```

Tip: In Java, TLS requires `SSLContext`, `KeyManagerFactory`, `TrustManagerFactory`, and an `SSLSocketFactory` or `SSLServerSocketFactory` — a ceremony that takes dozens of lines even for the common case. Go's `crypto/tls` collapses this to a `tls.Config` struct and a single function call. For HTTPS, `http.ListenAndServeTLS` reduces it further to two file paths.

Try It

Type this in and run it. It wires together the four headline APIs of this chapter — a `ServeMux` with method routing and a path wildcard, a logging middleware, JSON encoding to the response, and a JSON-decoding HTTP client — in one self-contained program. It uses `httptest.NewServer` so the whole round trip runs in a single process with no port to pick or `curl` to type.

```
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "net/http/httptest"
)

type Song struct {
    ID     string `json:"id"`
    Title  string `json:"title"`
    Artist string `json:"artist"`
}

var catalog = map[string]Song{
    "1": {ID: "1", Title: "Si Antes Te Hubiera Conocido", Artist: "Karol G"},
    "2": {ID: "2", Title: "Luna", Artist: "Feid"},
}

func logging(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        log.Printf("%s %s", r.Method, r.URL.Path)  // log every request
        next.ServeHTTP(w, r)                        // then call the handler
    })
}

func getSong(w http.ResponseWriter, r *http.Request) {
    song, ok := catalog[r.PathValue("id")]
    if !ok {
        http.Error(w, "no existe", http.StatusNotFound)
        return
    }
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(song)
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /songs/{id}/", getSong)

    srv := httptest.NewServer(logging(mux))  // start a real server on a random port
    defer srv.Close()

    resp, err := http.Get(srv.URL + "/songs/1/")
    if err != nil {
```

```
        log.Fatal(err)
    }
    defer resp.Body.Close()

    var s Song
    json.NewDecoder(resp.Body).Decode(&s)
    fmt.Printf("%d %s by %s\n", resp.StatusCode, s.Title, s.Artist)
    // 200 Si Antes Te Hubiera Conocido by Karol G
}
```

Try these modifications:

- Request `/songs/99/` instead of `/songs/1/` and confirm you get a `404` with the body `no existe`.
- Add a second middleware that sets a `X-Powered-By` header, and chain it inside `logging`.
- Add a `POST /songs/{id}/` route that uses `json.NewDecoder(r.Body)` to decode a `Song` from the request body and stores it in `catalog`.

Key Points

- `json.Marshal` encodes a Go value to `[]byte`; `json.Unmarshal` decodes `[]byte` into a pointer.
- Struct tags control JSON field names, `omitempty` (skip zero values), and `-` (always skip).
- Use the streaming `json.NewEncoder` and `json.NewDecoder` when reading from or writing to an `io.Reader`/`io.Writer` to avoid allocating the entire payload.
- `http.Handler` is the core interface; `http.HandlerFunc` adapts a plain function to satisfy it (Chapter 6 pattern).
- Go 1.22 `ServeMux` supports method routing (`GET /path/`) and path wildcards (`/songs/{id}/`) in the standard library — no third-party router needed for most services.
- Middleware is a function `func(http.Handler) http.Handler`; compose layers by wrapping.
- Always close `resp.Body` after an HTTP client response — `defer resp.Body.Close()` immediately after the error check.
- Reuse `http.Client` across requests; it manages a connection pool internally.
- `import _ "net/http/pprof"` registers live profiling endpoints on `DefaultServeMux`; use `go tool pprof -http` to analyze; never expose these on a public port.
- `encoding/xml` mirrors `encoding/json` exactly — same `Marshal`/`Unmarshal` API, same tag mechanism, different key (`xml` instead of `json`).
- `net.Dial` and `net.Listen` give raw TCP/UDP access; `net.Conn` implements `io.Reader` and `io.Writer`, so all `io` utilities compose with it.
- `http.ListenAndServeTLS` adds TLS to an HTTP server with two file paths; `tls.Config` controls certificate loading, minimum version, and custom CA trust.
- `tls.Dial` and `tls.Listen` mirror `net.Dial`/`net.Listen` for raw TLS; `tls.Conn` satisfies `net.Conn` so existing `io` code needs no changes.
- Never use `InsecureSkipVerify: true` in production — it disables all certificate validation.

Exercises

1. **Think about it:** In Java with Spring MVC or JAX-RS, you annotate a class method with `@GetMapping("/songs/{id}")` or `@GET @Path("/songs/{id}")` and the framework discovers handlers via reflection and classpath scanning. In Go, you call `mux.HandleFunc("GET /songs/{id}/", getSong)` explicitly in `main`. What are the tradeoffs of each approach? Consider startup time, debuggability, IDE navigation, and what happens when two handlers are registered for the same pattern.

2. **What does this print?**

```go
package main

import (
    "encoding/json"
    "fmt"
)

type Artist struct {
    Name    string `json:"name"`
    Country string `json:"country,omitempty"`
    Secret  string `json:"-"`
}

func main() {
    a := Artist{Name: "Chicane", Country: "", Secret: "UK"}
    data, _ := json.Marshal(a)
    fmt.Println(string(data))

    var b Artist
    json.Unmarshal([]byte(`{"name":"Darude","secret":"Finland"}`), &b)
    fmt.Printf("Name: %s, Secret: %q\n", b.Name, b.Secret)
}
```

3. **Calculation:** Consider the following `ServeMux` registration and the three incoming requests below. For each request, state which handler function is called. If no handler runs, give the HTTP error status the mux returns, and say whether the mux failed to match the path at all (`404 Not Found`) or matched the path pattern but not the request method (`405 Method Not Allowed`).

```go
mux := http.NewServeMux()
mux.HandleFunc("GET /tracks/",       listTracks)
mux.HandleFunc("GET /tracks/{id}/",  getTrack)
mux.HandleFunc("POST /tracks/",      createTrack)
```

   a. `GET /tracks/`
   b. `GET /tracks/42/`
   c. `DELETE /tracks/7/`

4. **Where is the bug?**

```go
package main

import (
    "fmt"
    "io"
    "net/http"
)

func fetchLyrics(url string) (string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return "", err
    }
    body, err := io.ReadAll(resp.Body)
    if err != nil {
        return "", err
    }
```

```
    return string(body), nil
}

func main() {
    lyrics, err := fetchLyrics("https://api.example.com/lyrics/sandstorm")
    if err != nil {
        fmt.Println("error:", err)
        return
    }
    fmt.Println(lyrics)
}
```

5. **Write a program:** Build a small in-memory HTTP server that manages a list of songs. Define a `Song` struct with fields `ID int`, `Title string`, and `Artist string`, all with appropriate `json` tags. Store songs in a package-level `map[int]Song`. Register two routes using a `http.NewServeMux()`:
   - `GET /songs/` returns all songs as a JSON array.
   - `GET /songs/{id}/` returns the single song with that ID, or HTTP 404 if not found. Pre-populate the map with two songs (use Darude or Chicane tracks). Start the server on `:8080`.