Chapter 2: The Engine — Backend Development in Go

Introduction: What You Will Learn

This is the longest and most important chapter. The backend is where actual logic happens. The frontend is just a pretty face; the backend is the brain.

By the end of this chapter, you will understand: 1. Basic Go syntax (variables, functions, structs) 2. How HTTP servers work in Go 3. Every line of our backend code 4. Why we use Mutex for concurrency 5. How to handle JSON data

---

Part 1: Go for Absolute Beginners

What is Go?

Go (also called Golang) is a programming language created by Google. It's known for: - **Simplicity**: Easy to learn - **Speed**: Compiles to machine code (fast like C) - **Concurrency**: Great for handling many requests at once

Basic Syntax

Variables

```go
// Method 1: Explicit type
var name string = "Alice"
var age int = 25

// Method 2: Type inference (Go figures it out)
name := "Alice"  // := means "create and assign"
age := 25
```

Functions

```go
// A function that takes two integers and returns one integer
func add(a int, b int) int {
    return a + b
}

// A function that returns two values
func divide(a, b int) (int, error) {
    if b == 0 {
        return 0, errors.New("cannot divide by zero")
    }
    return a / b, nil  // nil means "no error"
}
```

Structs (Like Classes)

```go
// Define a struct (a custom data type)
type User struct {
    ID       int
    Username string
    Photo    string
}

// Create an instance
user := User{
    ID:       1,
    Username: "alice",
    Photo:    "",
}

// Access fields
fmt.Println(user.Username)  // Prints: alice
```

Maps (Like Dictionaries)

```go
// Create a map from int to User
users := make(map[int]User)

// Add a user
users[1] = User{ID: 1, Username: "alice"}

// Get a user
user := users[1]

// Check if exists
user, exists := users[999]
if !exists {
    fmt.Println("User not found")
}
```

Slices (Like Arrays)

```go
// Create a slice (dynamic array)
messages := []string{"hello", "world"}

// Add to slice
messages = append(messages, "!")

// Loop through
for i, msg := range messages {
    fmt.Printf("Message %d: %s\n", i, msg)
}
```

Part 2: Project Structure Explained

Our backend is organized like this:

```
service/
  api/
     api.go          # Router setup (which URL goes where)
     handlers.go     # Handler functions (actual logic)
  cmd/
     api/
         main.go     # Entry point (starts the server)
  internal/
     database/
         database.go # In-memory database
  web/
      (frontend files)
```

Why This Structure?

Folder	Purpose	Why Separate?
`cmd/`	Entry points	A project can have multiple executables (CLI, server, etc.)
`api/`	HTTP logic	Separates "how to talk" from "what to do"
`internal/`	Private code	Go PREVENTS other projects from importing this
`web/`	Frontend files	Served as static files, not Go code

Oral Exam Defense: "I followed the Standard Go Project Layout. The `internal` package enforces encapsulation—database logic cannot be accessed by external packages. The `api` package is the public HTTP interface."

Part 3: main.go — The Entry Point

Let me explain every single line of `service/cmd/api/main.go`:

```go
package main
```

Line 1: Every Go file starts with `package`. The `main` package is special—it's the entry point of the program.

```go
import (
    "fmt"
    "log"
    "net/http"
```

Lines 3-6: We import packages we need: - `fmt`: Formatting and printing - `log`: Logging errors - `net/http`: The HTTP server

```
    "wasatext/service/api"
    "wasatext/service/internal/database"
)
```

Lines 7-8: We import OUR packages. The path matches the module name in `go.mod`.

```
func main() {
```

Line 10: The `main()` function runs when you start the program.

```
    // Initialize the database
    db, err := database.NewDatabase()
    if err != nil {
        log.Fatalf("Failed to create database: %v", err)
    }
```

Lines 11-15: 1. We create the database 2. The function returns `(db, err)` — two values 3. In Go, errors are explicit. We ALWAYS check if `err != nil` 4. `log.Fatalf` prints the error and EXITS the program

```
    // Create the router with all routes
    router := api.NewRouter(db)
```

Lines 17-18: We create an HTTP router. This is like a “switchboard” that says: - “If someone requests `/session`, run `doLogin`” - “If someone requests `/conversations`, run `getConversations`”

```
    // Start the server
    fmt.Println("Server starting on port 3000...")
    err = http.ListenAndServe(":3000", router)
    if err != nil {
        log.Fatalf("Server failed: %v", err)
    }
}
```

Lines 20-25: 1. Print a message so we know it’s starting 2. `http.ListenAndServe` starts a server on port 3000 3. The router handles incoming requests 4. If the server crashes, log the error

---

Part 4: database.go — The In-Memory Database

This is the most complex file. Let me explain it piece by piece.

The Data Structures

```
package database
```

Line 1: This package is named `database`. It lives in the `internal/database` folder.

```
import (
    "errors"
    "sync"
    "time"
)
```

Lines 3-7: - `errors`: For creating error messages - `sync`: For Mutex (thread safety) - `time`: For timestamps

```go
var (
    ErrUserNotFound         = errors.New("user not found")
    ErrConversationNotFound = errors.New("conversation not found")
    ErrMessageNotFound      = errors.New("message not found")
    ErrUnauthorized         = errors.New("unauthorized")
)
```

Lines 9-14: We define error constants. This is better than writing `errors.New("user not found")` every time because: 1. You can compare errors: `if err == ErrUserNotFound` 2. Typos are caught at compile time 3. Easy to change the message in one place

The User Type

```go
type User struct {
    ID       int    `json:"id"`
    Username string `json:"username"`
    Photo    string `json:"photo,omitempty"`
}
```

Lines 17-21: - `type User struct`: We're defining a custom type called User - `ID int`: An integer field named ID - `` `json:"id"` ``: A "tag" that tells JSON encoder to use "id" (lowercase) in JSON output - `omitempty`: If Photo is empty, don't include it in JSON

Without tags:

```json
{"ID": 1, "Username": "alice", "Photo": ""}
```

With tags:

```json
{"id": 1, "username": "alice"}
```

The Database Structure

```go
type AppDatabase struct {
    users         map[int]User
    usersByName   map[string]int
    conversations map[int]Conversation
    messages      map[int][]Message

    lastUserID    int
    lastConvID    int
    lastMessageID int

    mu sync.RWMutex
}
```

Lines 44-57: This is our "database". Let me explain each field:

Field	Type	Purpose
`users`	`map[int]User`	Stores users by ID. `users[1]` gives User with ID 1
`usersByName`	`map[string]int`	Index for login. `usersByName["alice"]` gives Alice's ID
`conversations`	`map[int]Conversation`	Stores conversations by ID
`messages`	`map[int][]Message`	Messages grouped by conversation. `messages[5]` = all messages in conv 5
`lastUserID`	`int`	Counter for generating unique IDs
`mu`	`sync.RWMutex`	**THE LOCK** (explained in Part 5)

Creating the Database

```go
func NewDatabase() (*AppDatabase, error) {
    return &AppDatabase{
        users:         make(map[int]User),
        usersByName:   make(map[string]int),
        conversations: make(map[int]Conversation),
        messages:      make(map[int][]Message),
        lastUserID:    0,
        lastConvID:    0,
        lastMessageID: 0,
    }, nil
}
```

Lines 60-71: - `func NewDatabase()`: A function to create a new database - `*AppDatabase`: Returns a POINTER (memory address) to the database - `make(map[...])`: Creates an empty map - `nil`: No error, everything is fine

---

Part 5: Understanding the Mutex (Critical for Oral Exam!)

The Problem: Race Conditions

Imagine this scenario:

1. User A sends a message (needs to increment `lastMessageID`: 10 □ 11)
2. User B sends a message at the SAME TIME (also reads `lastMessageID`: still 10!)
3. Both messages get ID 11!
4. **BUG**: Duplicate IDs, data corruption

This is called a **race condition**.

The Solution: Locks

A Mutex (Mutual Exclusion) is like a bathroom key: - Only one person can hold the key at a time - Everyone else waits outside - When you're done, you return the key

```
func (db *AppDatabase) LoginUser(username string) (User, error) {
    db.mu.Lock()            //  Grab the key. Everyone else WAITS.
    defer db.mu.Unlock()    //  Promise to return the key when done.

    // ... do stuff safely ...
}
```

RWMutex: A Smarter Lock

A `sync.RWMutex` has TWO modes:

Method	Who Can Hold It	Use Case
`Lock()`	ONE writer only	Creating, updating, deleting
`RLock()`	MANY readers	Just reading data

Why? - Reading doesn't change data - 100 people can read a book simultaneously - But only one person should write in it

```
// WRITING - Exclusive lock
func (db *AppDatabase) SendMessage(...) {
    db.mu.Lock()            // Only I can access
    defer db.mu.Unlock()
    // ... add message ...
}

// READING - Shared lock
func (db *AppDatabase) GetMessages(...) {
    db.mu.RLock()           // Others can also read
    defer db.mu.RUnlock()
    // ... return messages ...
}
```

What is `defer`?

`defer` says: "Run this line AFTER the function ends, no matter what."

```
func example() {
    db.mu.Lock()
    defer db.mu.Unlock()  // Will run at the end

    if somethingBad {
        return  // defer still runs! Key is returned.
    }
    // ... more code ...
}  // defer runs here too!
```

Without `defer`, if you forgot to unlock (or returned early), the program would **deadlock** — everyone waiting for a key that's never returned.

Oral Exam Defense: "I use `sync.RWMutex` because Go maps are not thread-safe. `RWMutex` allows multiple concurrent readers but exclusive access for writers. The `defer` keyword guarantees the lock is always released, preventing deadlocks."

Part 6: handlers.go — The HTTP Handlers

This file contains all the functions that handle HTTP requests.

The Structure

Every handler follows this pattern:

```go
func (api *ApiServer) handlerName(w http.ResponseWriter, r *http.Request) {
    // 1. Authenticate (who is calling?)
    // 2. Parse input (what are they sending?)
    // 3. Validate (is the input valid?)
    // 4. Execute (call the database)
    // 5. Respond (send back JSON)
}
```

Let me explain the key parts:

`func (api *ApiServer)` — Method Receiver

```go
func (api *ApiServer) doLogin(w http.ResponseWriter, r *http.Request)
```

This means: - `doLogin` is a METHOD attached to `ApiServer` - `api` is like `this` or `self` in other languages - We can access `api.db` to talk to the database

`w http.ResponseWriter` — The Response

`w` is how we SEND data back to the client:

```go
w.WriteHeader(http.StatusOK)       // Set status code to 200
w.Write([]byte("Hello, World!"))   // Send some text
```

`r *http.Request` — The Request

`r` contains everything from the client:

```go
// Get a header
token := r.Header.Get("Authorization")
```

```go
// Get URL parameter (/conversations/{id})
id := r.PathValue("id")

// Get query parameter (/users?q=alice)
query := r.URL.Query().Get("q")

// Read the body (JSON data)
json.NewDecoder(r.Body).Decode(&body)
```

Complete Handler Walkthrough: doLogin

```go
func (api *ApiServer) doLogin(w http.ResponseWriter, r *http.Request) {
```

Line 1: Function signature. It's a method on `ApiServer`.

```go
    var body struct {
        Username string `json:"username"`
    }
```

Lines 2-4: Define the expected JSON structure. We expect `{"username": "alice"}`.

```go
    if err := json.NewDecoder(r.Body).Decode(&body); err != nil {
        errorResponse(w, http.StatusBadRequest, "Invalid JSON")
        return
    }
```

Lines 5-8: 1. `json.NewDecoder(r.Body)`: Create a decoder that reads from the request body 2. `.Decode(&body)`: Fill the `body` variable with the JSON data 3. `&body`: The `&` means "give the decoder the memory address so it can write to it" 4. If decoding fails, send a 400 Bad Request error and STOP

```go
    if len(body.Username) < 3 {
        errorResponse(w, http.StatusBadRequest, "Username must be at least 3 characters")
        return
    }
```

Lines 9-12: Validation. The username must be at least 3 characters.

```go
    user, err := api.db.LoginUser(body.Username)
    if err != nil {
        errorResponse(w, http.StatusInternalServerError, err.Error())
        return
    }
```

Lines 13-17: 1. Call the database's `LoginUser` function 2. It either finds or creates the user 3. If something goes wrong, send a 500 error

```go
    jsonResponse(w, http.StatusCreated, map[string]interface{}{
        "id":    user.ID,
        "token": strconv.Itoa(user.ID),
    })
}
```

Lines 18-21: 1. Send a 201 Created response 2. The body is a JSON object with `id` and `token` 3. `strconv.Itoa`: Convert integer to string (ID □ "token")

Complete Handler Walkthrough: sendMessage

```go
func (api *ApiServer) sendMessage(w http.ResponseWriter, r *http.Request) {
    // Step 1: Authenticate
    userID, err := api.authenticate(r)
    if err != nil {
        errorResponse(w, http.StatusUnauthorized, "Not logged in")
        return
    }
```

Authentication: Extract the user ID from the `Authorization: Bearer 123` header.

```go
    // Step 2: Get conversation ID from URL
    convIDStr := r.PathValue("conversationId")
    convID, err := strconv.Atoi(convIDStr)
    if err != nil {
        errorResponse(w, http.StatusBadRequest, "Invalid conversation ID")
        return
    }
```

Parse URL: Get `/conversations/5/messages` □ `convID = 5`.

```go
    // Step 3: Parse body
    var body struct {
        Content  string `json:"content"`
        Photo    string `json:"photo"`
        ReplyTo  *int   `json:"reply_to_message_id"`
    }
    json.NewDecoder(r.Body).Decode(&body)
```

Parse Body: Get the message content. Note `*int` means "optional integer" (can be omitted).

```go
    // Step 4: Validate
    if body.Content == "" && body.Photo == "" {
        errorResponse(w, http.StatusBadRequest, "Message cannot be empty")
        return
    }
```

Validate: Must have text or photo.

```go
    // Step 5: Execute
    msg, err := api.db.SendMessage(convID, userID, body.Content, body.Photo, body.ReplyTo, nil
    if err != nil {
        // Handle specific errors
        if err == database.ErrConversationNotFound {
            errorResponse(w, http.StatusNotFound, "Conversation not found")
        } else {
            errorResponse(w, http.StatusInternalServerError, err.Error())
        }
        return
    }
```

Execute: Call the database. Handle different errors differently.

```
    // Step 6: Respond
    jsonResponse(w, http.StatusCreated, msg)
}
```

Respond: Return the created message as JSON.

Part 7: api.go — The Router

The router connects URLs to handlers.

```
func NewRouter(db *database.AppDatabase) http.Handler {
    api := &ApiServer{db: db}

    mux := http.NewServeMux()
```

Lines 1-4: 1. Create an `ApiServer` that holds the database reference 2. Create a “multiplexer” (mux) — a fancy word for router

```
    // Auth & Users
    mux.HandleFunc("POST /session", api.doLogin)
    mux.HandleFunc("PUT /users/{userId}/name", api.setMyUserName)
    mux.HandleFunc("PUT /users/{userId}/photo", api.setMyPhoto)
    mux.HandleFunc("GET /users", api.searchUsers)
```

Lines 6-10: Register routes. Format: `"METHOD /path"`.

The pattern `{userId}` means it’s a variable. Go 1.22+ can extract it with `r.PathValue("userId")`.

```
    // Conversations
    mux.HandleFunc("GET /conversations", api.getMyConversations)
    mux.HandleFunc("POST /groups", api.createConversation)
    mux.HandleFunc("GET /conversations/{conversationId}", api.getConversation)

    // Messages
    mux.HandleFunc("GET /conversations/{conversationId}/messages", api.getConversationMessages
    mux.HandleFunc("POST /conversations/{conversationId}/messages", api.sendMessage)
    mux.HandleFunc("DELETE /conversations/{conversationId}/messages/{messageId}", api.deleteMes

    return enableCORS(mux)
}
```

Final Line: Wrap the router with CORS middleware (explained below).

CORS Middleware

```
func enableCORS(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Set CORS headers
        w.Header().Set("Access-Control-Allow-Origin", "*")
        w.Header().Set("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS")
        w.Header().Set("Access-Control-Allow-Headers", "Content-Type, Authorization")
        w.Header().Set("Access-Control-Max-Age", "1")

        // Handle preflight
        if r.Method == "OPTIONS" {
            w.WriteHeader(http.StatusOK)
            return
        }

        next.ServeHTTP(w, r)
    })
}
```

What is CORS? Cross-Origin Resource Sharing. Browsers block requests from `localhost:8081` (frontend) to `localhost:3000` (backend) by default.

CORS headers tell the browser: "It's OK, I trust this origin."

- `Access-Control-Allow-Origin: *`: Allow requests from anywhere
- `Access-Control-Allow-Methods`: Allow these HTTP methods
- `Access-Control-Allow-Headers`: Allow these headers
- `Access-Control-Max-Age: 1`: Cache this permission for 1 second (strict requirement from PDF)

Oral Exam Defense: "CORS is a browser security mechanism. I implemented middleware that adds the required headers to every response. The Max-Age of 1 second ensures permission changes propagate immediately."

Summary Checklist

Before moving to Chapter 3, make sure you understand:

- ☐ Basic Go syntax (structs, maps, slices, functions)
- ☐ What a Mutex does and why we need it
- ☐ The difference between `Lock()` and `RLock()`
- ☐ What `defer` means
- ☐ How to read and parse HTTP requests
- ☐ How to send HTTP responses
- ☐ What CORS is and why we need it
- ☐ The handler pattern (authenticate → parse → validate → execute → respond)

Oral Exam Questions

Q: Why Go instead of Node.js or Python? “Go compiles to a single binary—no runtime dependencies. It has excellent performance due to native compilation. The standard library includes a production-ready HTTP server. Goroutines make concurrent request handling simple.”

Q: Explain sync.RWMutex. “RWMutex is a reader-writer lock. Multiple goroutines can hold read locks simultaneously, but write locks are exclusive. This optimizes for read-heavy workloads like fetching messages while ensuring data consistency when writing.”

Q: What happens if you forget to unlock? “The program deadlocks. All subsequent requests wait forever for the lock. That’s why I use `defer db.mu.Unlock()` immediately after locking—it guarantees the unlock happens even if the function returns early or panics.”