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 main backend code
4. **How we use SQLite** (unlike the in-memory example, we use a real database file)
5. How to prevent SQL Injection

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

```
// Method 1: Explicit type
var name string = "Alice"

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

Functions

```
// 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 (Result and Error)
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)

```
// Define a custom data type
type User struct {
    ID       int
    Username string
}

// Create an instance
me := User{ID: 1, Username: "Alice"}
fmt.Println(me.Username) // Prints: Alice
```

Part 2: Project Structure Explained

Our backend is organized like this:

```
wasatext/
├── cmd/
│   └── webapi/
│       └── main.go       # Entry point (Starts the server)
├── service/
│   ├── api/
│   │   ├── api.go        # Router setup & Middleware
│   │   └── messages.go   # Handlers (The logic)
│   └── database/
│       └── database.go   # Database connection & SQL queries
└── go.mod                # Dependency file
```

Why This Structure?

- **`cmd/`**: Standard place for "Main" files. Separation allows multiple binaries (e.g., a server and a CLI tool).
- **`service/api`**: Contains the **HTTP Layer** (Decodes JSON, checks headers).
- **`service/database`**: Contains the **Data Layer** (SQL Queries).

Oral Exam Defense:

> *"I followed the Standard Go Project Layout. I separated the `api` (HTTP logic) from the `database` (SQL logic) to ensure Separation of Concerns. This makes the code easier to test and maintain."*

Part 3: `main.go` — The Entry Point

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

```
func main() {
    // 1. Get the port (default to 3000)
    port := os.Getenv("PORT")
    if port == "" {
        port = "3000"
    }

    // 2. Initialize the Database
    // We open "wasatext.db". If it doesn't exist, it is created.
    db, err := database.New("wasatext.db")
    if err != nil {
        log.Fatalf("Error initializing database: %v", err)
    }
    defer db.Close() // Ensure database closes when program stops

    // 3. Create the API Handler
    // We inject the database into the API.
    apiHandler := api.New(db)

    // 4. Create the Router
    router := api.NewRouter(apiHandler)

    // 5. Start the Server
    http.ListenAndServe(":"+port, router)
}
```

Key Concept: Dependency Injection Notice `api.New(db)`. We create the database *outside* and pass it *in*. This is cleaner than the API creating its own database connection.

Part 4: `database.go` — The SQLite Database

We use **SQLite**. It's a file-based SQL database.

The `createTables` Function

When the server starts, we run this SQL to ensure our tables exist:

```
func createTables(db *sql.DB) error {
    _, err := db.Exec(`
        CREATE TABLE IF NOT EXISTS users (
            id TEXT PRIMARY KEY,
            name TEXT UNIQUE NOT NULL,
            photo BLOB
        );

        CREATE TABLE IF NOT EXISTS messages (
            id TEXT PRIMARY KEY,
            content TEXT,
            sender_id TEXT,
            -- ... other fields
            FOREIGN KEY (sender_id) REFERENCES users(id)
        );
    `)
    return err
}
```

SQL Injection Prevention (CRITICAL)

The Problem: If you do this: `db.Exec("SELECT * FROM users WHERE name = " + userInput)` A hacker can send: `userInput = "a; DROP TABLE users;"` This deletes your database.

The Solution: We use **Parameterized Queries** (`?` placeholders).

```
// Secure way
db.QueryRow("SELECT id FROM users WHERE name = ?", name)
```

The database treats `name` strictly as *data*, never as executable *code*.

Part 5: `handlers.go` — The Logic

The handlers (like `SendMessage` in `service/api/messages.go`) follow a specific **7-Step Pattern**:

```
func (h *Handler) SendMessage(w http.ResponseWriter, r *http.Request) {
    // 1. Authenticate (Check Header)
    userID := getUserIDFromAuth(r)
    if userID == "" {
        http.Error(w, "Unauthorized", http.StatusUnauthorized)
        return
    }

    // 2. Parse URL Parameters
    vars := mux.Vars(r)
    conversationID := vars["conversationId"]

    // 3. Parse Body (JSON)
    var req SendMessageRequest
    json.NewDecoder(r.Body).Decode(&req)

    // 4. Validate Input
    if req.Content == "" {
        http.Error(w, "Message cannot be empty", http.StatusBadRequest)
        return
    }

    // 5. Execute Logic (Call Database)
    msg, err := h.db.CreateMessage(conversationID, userID, req.Content, ...)

    // 6. Respond (Success)
    w.WriteHeader(http.StatusCreated) // 201 Created
    json.NewEncoder(w).Encode(msg)    // Send back JSON
}
```

Part 6: `api.go` — The Router & CORS

The Router (`gorilla/mux`)

We use a library called `gorilla/mux` because it handles parameters (like `{id}`) easily.

```
r := mux.NewRouter()
r.HandleFunc("/session", h.DoLogin).Methods("POST")
r.HandleFunc("/conversations/{id}/messages", h.SendMessage).Methods("POST")
```

CORS Middleware

What is CORS? (Cross-Origin Resource Sharing) Browsers block requests from one domain (e.g., `localhost:8080`) to another (`localhost:3000`) by default.

We need to tell the browser: "It's okay, allow this."

```
w.Header().Set("Access-Control-Allow-Origin", "*")
w.Header().Set("Access-Control-Allow-Methods", "GET, POST, OPTIONS, ...")
```

Oral Exam Defense:

> *"I implemented CORS middleware because the frontend serves from a different port than the backend. The `Allow-Origin: *` header tells the browser to permit these cross-origin requests."*

Summary Checklist

Before moving to Chapter 3, ensure you understand:

- [ ] Basic Go syntax (`type`, `func`, `if err != nil`)
- [ ] Why we separate `cmd` and `service`
- [ ] How `main.go` wires everything together
- [ ] Why we use SQLite (`sql.Open`) instead of a simple Go map
- [ ] **SQL Injection** and how `?` prevents it
- [ ] The "Handler Pattern" (Auth -> Parse -> DB -> Respond)
- [ ] What CORS is and why we need it

Oral Exam Questions

Q: Why Go? "Go compiles to a single binary, is strongly typed, and handles HTTP natively. It's built for systems like this."

Q: Why not use an ORM (like GORM)? "I chose to use raw SQL with `database/sql` to demonstrate my understanding of relational databases and SQL syntax. ORMs hide complexity, but for this project, understanding the underlying data manipulation is key."

Q: What happens if you don't close the database? "Resource leaks. Eventually, the operating system will run out of file descriptors, and the server will crash. That's why we use `defer db.Close()`."