Chapter 1: Requirements Analysis

1.1 Introduction

This chapter explains **how we started**. In any software project, before writing code, you must understand **what** to build. Our source of truth was the **WASAText PDF**.

1.2 The "Single Source of Truth" Rule

For this exam project, strictly adhering to the PDF is critical. This prevents "feature creep" (adding unnecessary features) and ensures you meet the professor's grading criteria.

Key Rule Used:

> *"If it's not in the PDF, I don't do it."*

1.3 Key Functional Requirements

We broke down the PDF into specific features. Here is what we extracted and **why**:

A. The Login System

- **Requirement:** "A user can log in simply by entering their username."
- **Technical Detail:** We use **Bearer Authentication**.
  - **How it works:**
    1. Client sends `username`.
    2. Server returns an `identifier` (e.g., `abc-123`).
    3. Client sends this `identifier` in the header `Authorization: Bearer abc-123` for all future requests.
  - **Simplicity:** No passwords, no sessions, no cookies. Just a database lookup.

B. Conversations

- **Requirement:** List sorted in **reverse chronological order** (newest first).
- **Details:** Must show the last message snippet, time, and photos.

C. Messaging

- **Types:** Text and Photos/GIFs.
- **Status Checkmarks:**
  - ✓ **(One Check):** Received by server (or delivered to recipient's list).

- ✓✓ **(Two Checks):** Read by recipient (they opened the chat).
- *Implementation Note:* In our simple backend, we mark messages as `received` immediately when created, and `read` when the user requests `GET /conversations/{id}`.

D. Groups

- **Creation:** Anyone can create a group.
- **Membership:** Users *cannot* join groups themselves; they must be added by a member.
- **Visibility:** You can only see groups you belong to.

1.4 Exam Questions & Answers

Q1: Why did you choose SQLite for the database?

Answer: The PDF didn't specify a database engine, but SQLite is the standard choice for Go learning projects because:

1. It requires no separate server process (it's just a file).
2. It supports standard SQL, fulfilling the requirement for a relational database structure.
3. It simplifies the architecture for the exam demo.

Q2: Explain the "Simplified Login" flow. Is it secure?

Answer:

- **Flow:** The user provides a username. If it exists, we log them in; if not, we create it. The server returns a UUID. The client stores this UUID and sends it in the `Authorization` header.
- **Security: No, it is not secure** for production. Anyone knowing the username could potentially log in if we didn't use UUIDs. In this project, the "identifier" acts like a secret token. Real apps use passwords or OAuth (Google/Facebook login). The PDF explicitly stated to use this simplified flow to avoid complexity.

Q3: What is "Bearer Authentication"?

Answer: It's an HTTP authentication scheme. "Bearer" means "give access to the bearer of this token". It's like a bus ticket – whoever holds the ticket gets on the bus. In our app, the "token" is the User ID returned by the login endpoint.

Q4: How do you handle "Reverse Chronological Order"?

Answer: In the database query (SQL), we use `ORDER BY timestamp DESC`.

- **DESC (Descending):** Newest dates first (e.g., 2024... then 2023...).
- **ASC (Ascending):** Oldest dates first. We apply this to both the Conversation List and the Messages within a chat.

Q5: Why didn't you implement features like "Typing indicators" or "Online status"?

Answer: They were **not specified in the PDF**. Following the strict scope rules, I implemented only what was requested to ensure the project met the exact functional design specifications without unnecessary complexity.

Chapter 2: OpenAPI Design

2.1 What is OpenAPI?

OpenAPI is a **specification** (a standard format) for defining APIs. Think of it as a **contract** between the Frontend and the Backend.

- **Backend Developer says:** "I promise my server will accept *this* data and return *that* data."
- **Frontend Developer says:** "Okay, I will write my code to send *this* and expect *that*."

In this project, we wrote the contract (`openapi.yaml`) **before** writing the code. This is called **"API-First Design"**.

2.2 The 14 Key APIs

The PDF required specific `operationId`s. Here is what each one does:

authentication

1. `doLogin`: **POST /session**
    - Takes `{ name: "Maria" }`
    - Returns `{ identifier: "123-abc" }`

users

2. `setMyUserName`: **PUT /users//username**
    - Updates your display name.
3. `setMyPhoto`: **PUT /users//photo**
    - Uploads a profile picture.
    - *Note:* Uses `image/*` content type (binary data), not JSON.

conversations

4. `getMyConversations`: **GET /conversations**
    - Returns the list of chats for the main screen.
5. `getConversation`: **GET /conversations/**
    - Returns the full chat history with messages.

messages

6. `sendMessage`: **POST /conversations//messages**
    - Sends text or photo.
7. `forwardMessage`: **POST .../forward**
    - Copies a message to another chat.
8. `deleteMessage`: **DELETE .../messages/**
    - Removes a message (only if you sent it).

reactions

9. `commentMessage`: **POST .../comments**
    - Adds an emoji reaction (e.g., "□").
10. `uncommentMessage`: **DELETE .../comments**
    - Removes the reaction.

groups

11. `addToGroup`: **POST /groups//members**
    - Adds a user (by their ID) to the group.
12. `leaveGroup`: **DELETE /groups//members/me**
    - Removes *yourself* from the group.
13. `setGroupName`: **PUT /groups//name**
14. `setGroupPhoto`: **PUT /groups//photo**

2.3 Exam Questions & Answers

Q1: Why do we use `PUT` for updating the username but `POST` for sending a message?

Answer:

- **PUT** is roughly for "Update" or "Replace". When we set a username, we are replacing the old one. Ideally, `PUT` is idempotent (doing it twice has the same result as doing it once).
- **POST** is for "Create". Every time we call `sendMessage`, we create a *new* message. If we called it twice, we'd send two messages.

Q2: What does the `tags` field do in OpenAPI?

Answer: `tags` are used to group related APIs together in documentation. For example, we tagged `sendMessage` and `deleteMessage` under `["message"]` so they appear together in generated docs.

Q3: How did you define the Photo upload in OpenAPI?

Answer: We defined the request body content type as `image/*` (or `multipart/form-data` for messages) and the schema type as `string` with `format: binary`. This tells clients to send raw file bytes, not JSON text.

Q4: What is the difference between `GET /conversations` and `GET /conversations/

Answer:

- `GET /conversations` returns a **summary list** (previews only, showing the last message). It's light and fast for the sidebar.
- `GET /conversations/{id}` returns the **full details** (all messages, members). It's heavier, so we only fetch it when the user clicks a specific chat.

Q5: If I added a new feature, would I change the OpenAPI file first?

Answer: Yes. In API-First Design, you always update the contract (`openapi.yaml`) first, then implement the changes in the Backend (Go) and Frontend (JS). This keeps everyone in sync.

Chapter 3: Backend Implementation (Go)

3.1 Project Structure

We organized the code following the "Standard Go Project Layout":

```
cmd/webapi/main.go   <-- Entry Point (Starts everything)
service/
  api/               <-- HTTP Handlers (The "Waiters" taking orders)
  database/          <-- Database Logic (The "Chefs" cooking data)
```

3.2 The Entry Point (`main.go`)

This file does three things:

1. **Opens the Database:** `database.New(...)`
2. **Creates the Router:** `api.NewRouter(...)`
3. **Starts the Server:** `http.ListenAndServe(...)`

3.3 The Router (`service/api/api.go`)

We used `gorilla/mux`. A router maps URLs to functions.

- **Example Code:**

  ```
  r.HandleFunc("/session", h.DoLogin).Methods("POST")
  ```

  This means: "If a user POSTs to `/session`, verify it and run `DoLogin`."

CORS Middleware (Crucial PDF Requirement)

We wrote a manual middleware `corsMiddleware` to:

1. Intercept every request.
2. Add headers: `Access-Control-Allow-Origin: *` (Allow existing).
3. Handle `OPTIONS` requests (Pre-flight checks from browsers).

- **Why?** Browsers block requests from one domain (frontend) to another (backend) unless these headers exist.

3.4 Database Logic (`service/database/`)

We used **SQL** directly.

Example: Finding a User (`CreateUser`)

```
// We assume 'name' is unique.
// INSERT OR IGNORE means: "Try to add this user. If name exists, do nothing."
_, err := db.Exec("INSERT OR IGNORE INTO users (id, name) VALUES (?, ?)", id, name)
```

Then we select the ID back to return it.

Example: Getting Conversations

Most complex query. We need to join tables.

```
SELECT c.id, u.name, m.content
FROM conversations c
JOIN users u ON ...
LEFT JOIN messages m ON ...
ORDER BY m.timestamp DESC
```

This fetches the chat info AND the user info AND the last message all at once.

3.5 Exam Questions & Answers

Q1: Explain how Go handles concurrent requests.

Answer: Go's `http.Server` automatically starts a new **Goroutine** for every incoming request. A goroutine is a lightweight thread. This means 100 users can hit `/session` at the exact same instant, and Go will handle them all in parallel without us writing extra code.

Q2: Why did you use `gorilla/mux` instead of the standard library?

Answer: The standard Go library (before Go 1.22) had very weak routing capabilities (couldn't easily extract variables like `/users/{id}`). `gorilla/mux` makes it easy to get `id` using `mux.Vars(r)["id"]`.

Q3: What happens if the database is locked?

Answer: SQLite is a file-based DB and only allows one **writer** at a time (but many readers). If two users try to register instantly, one might be blocked briefly. For a bachelor project scope, this is acceptable. In production, we'd use PostgreSQL.

Q4: Explain the `defer db.Close()` in `main.go`.

Answer: `defer` schedules a function call to run **when the current function exits**. So `defer db.Close()` ensures that even if the server crashes or we stop it, the database connection is closed properly, preventing file corruption or resource leaks.

Q5: What is `json.NewEncoder(w).Encode(resp)` doing?

Answer:

1. It takes a Go struct (like `User{Name: "Bob"}`).
2. It converts it into a JSON string (`{"name": "Bob"}`).
3. It writes that string directly to the HTTP Response Writer (`w`), sending it to the client. It's a one-line way to send JSON responses.

Chapter 4: Frontend Implementation (Vue.js)

4.1 Why Vue.js?

For this project, we chose **Vue.js using CDN** (Content Delivery Network).

- **Simplicity:** No `npm`, no `webpack`, no `build` tools. Just one `index.html` file.
- **Reactivity:** We change a variable in data (`this.messages = [...]`), and the HTML updates automatically. Doing this in plain JS would require complex DOM manipulation code (`document.createElement`, etc.).

4.2 Application State

The entire app state is stored in one object:

```
data() {
    return {
        user: { ... },          // Who am I?
        conversations: [],      // My list of chats
        activeConv: null,       // Which chat is open?
        messages: [],           // Messages in open chat
        pollInterval: null      // Timer ID
    }
}
```

4.3 Key Components

4.3.1 Login

- Checks if `user.identifier` is null.
- If null, shows Login Box.
- Calls `api.login()`, gets ID, saves it.
- Then shows the Main UI.

4.3.2 The Polling Mechanism (Real-time Simulation)

Since we didn't use WebSockets (which are complex), how do we see new messages? **Polling.** We set an interval to run every 2 seconds:

```
setInterval(() => {
    this.refreshConversations(); // Check for new chats/previews
    if (this.activeConv) this.refreshMessages(); // Check for new msgs
}, 2000);
```

This asks the server "Anything new?" constantly.

4.3.3 Conditional Classes

We use Vue's `:class` binding for message bubbles:

```
<div :class="['message-bubble', msg.senderId === user.identifier ? 'message-out' : 'message-in']">
```

- If I sent it -> `message-out` (Green, Right aligned).
- If you sent it -> `message-in` (White, Left aligned).

4.4 CSS (Styling)

We used "Vanilla CSS" (standard CSS).

- **Flexbox:** Used everywhere (`display: flex`) to layout the Sidebar vs Chat Area side-by-side.
- **Variables:** `:root { --primary-color: ... }` to make changing theme colors easy.

4.5 Exam Questions & Answers

Q1: What is the "Virtual DOM" in Vue.js?

Answer: Vue keeps a copy of the HTML structure in memory. When data changes, it compares the new Virtual DOM with the old one, calculates the minimal changes needed, and updates the real browser DOM. This makes updates very fast.

Q2: Why use `axios` instead of `fetch`?

Answer: `axios` is a library that makes HTTP requests easier:

1. It automatically converts JSON responses (no need for `.json()`).
2. It handles errors better (throws error on 4xx/5xx status codes, unlike fetch).
3. It works well with older browsers.

Q3: How do you secure the pages if it's a Single Page App (SPA)?

Answer: We check `v-if="!user.identifier"` in the HTML template. If the variable is null, the "Main App" div isn't even rendered. However, a smart user could change the variable in the console. True security happens on the **Backend**: every API call checks the `Authorization` header. Even if they hack the UI, they can't get data without a valid token.

Q4: Explain the `v-for` directive.

Answer: `v-for="msg in messages"` acts like a loop. It repeats the HTML element for every item in the array. We must provide a `:key` (like `msg.messageId`) so Vue can track which specific items change order or are deleted.

Q5: What would be better than Polling?

Answer: WebSockets. WebSockets open a permanent 2-way connection. The server can "push" a new message instantly. Polling wastes battery and bandwidth by asking "Any news? No. Any news? No." repeatedly. But Polling is much simpler to implement for an exam.

Chapter 5: Docker Deployment

5.1 What is Docker?

Docker packages your application and everything it needs (OS libraries, files, environment settings) into a "Container".

- **Result:** "It works on my machine" becomes "It works on ANY machine."

5.2 The Dockerfile Explained

We used a **Multi-Stage Build**. This is an industry best practice to keep image size small.

Stage 1: The Builder (`FROM golang:1.21.6-alpine`)

- This stage has the Go compiler (which is heavy).
- We copy `go.mod`, download deps, and run `go build`.
- Result: A binary file named `wasatext`.

Stage 2: The Runner (`FROM alpine:3.19`)

- This stage is a tiny Linux OS (5MB). It has NO Go compiler.
- We `COPY --from=builder` *only* the binary and the `webui` folder.
- Result: A lightweight image containing only what runs.

5.3 Key Commands

- `docker build -t wasatext .`
  - Reads `Dockerfile`, runs steps, tags image as `wasatext`.
- `docker run -p 3000:3000 wasatext`
  - Starts container.
  - Maps **Host Port 3000** (your Mac) to **Container Port 3000** (inside Linux).

5.4 Exam Questions & Answers

Q1: Why did you use `alpine`?

Answer: Alpine Linux is extremely small (around 5MB). Standard Linux images (like Ubuntu) are 100MB+. Using Alpine makes our final docker image much smaller, faster to download, and safer (less software = fewer security holes).

Q2: What does `EXPOSE 3000` do?

Answer: It is **documentation only**. It tells the person reading the Dockerfile "Hey, this app listens on port 3000." It does *not* actually publish the port. You still need `-p 3000:3000` in the run command to do that.

Q3: Why copy `go.mod` before the source code?

Answer: Caching. Docker caches layers.

1. If we change `main.go` but not `go.mod`, Docker reuses the "downloaded dependencies" layer.
2. If we copied everything at once, *any* code change would force Docker to re-download all Go libraries, which is slow.

Q4: If I close the Docker container, is my data saved?

Answer: No. By default, files inside a container (like our `wasatext.db`) are ephemeral (temporary). If you delete the container, the DB is gone. To persist data, we would need to use a **Docker Volume** (e.g., `-v $(pwd)/data:/app/data`).

Q5: What is the difference between an Image and a Container?

Answer:

- **Image:** The recipe/blueprint (The read-only template). Like a Class in OOP.
- **Container:** The running instance. Like an Object in OOP. You can run 10 containers from 1 image.