Chapter 0: Foundations — Understanding Web Applications

Introduction: What You Will Learn

Before you write a single line of code, you need to understand *what* you are building and *why* certain decisions are made. This chapter is your foundation. If you skip this, nothing else will make sense.

By the end of this chapter, you will understand:

1. What a "web application" actually is
2. How the internet works (simplified)
3. What "Client-Server Architecture" means
4. What REST APIs are and why we use them
5. What authentication means (and why we're simplifying it)

Part 1: What is a Web Application?

The Simple Answer

A web application is a program that runs in two places:

1. **Your browser** (called the "Frontend" or "Client")
2. **A computer somewhere else** (called the "Backend" or "Server")

When you use WASAText:

- **The Frontend** is what you see: the chat bubbles, the login box, the list of users.
- **The Backend** is what you don't see: the computer storing all your messages and photos.

Why Split It?

Imagine you're building a **Library**:

- **The Frontend** is the reception desk where people ask for books.
- **The Backend** is the secure warehouse where books are actually stored.

You split them because:

1. **Security:** You don't want random people walking into your warehouse.
2. **Scalability:** You can have many reception desks but one central warehouse.
3. **Separation of concerns:** The receptionist doesn't need to know how to repair a book binding; they just need to find it.

Part 2: How the Internet Works (Simplified)

The Request-Response Cycle

When you type `google.com` in your browser:

1. **Your browser** sends a **Request** to Google's servers.
    - "Hey Google, give me your homepage"
2. **Google's server** processes this request.
3. **Google's server** sends back a **Response**.
    - "Here's the HTML for my homepage"
4. **Your browser** displays it.

This is called the **Request-Response Cycle**. EVERY interaction on the web follows this pattern.

```
[Your Browser] ----REQUEST----> [Server]
[Your Browser] <---RESPONSE---- [Server]
```

HTTP: The Language of the Web

HTTP (HyperText Transfer Protocol) is the "language" browsers and servers speak.

An HTTP Request has:

- **Method:** What you want to do (GET, POST, PUT, DELETE)
- **URL:** Where you want to do it (`/messages`)
- **Headers:** Extra information (like your login token)
- **Body:** Data you're sending (like the text of a message)

An HTTP Response has:

- **Status Code:** Did it work? (200 = Yes, 404 = Not Found, 500 = Server Error)
- **Headers:** Extra information
- **Body:** The data you asked for

Part 3: Client-Server Architecture in WASAText

Our Architecture

```
     YOUR COMPUTER
         |
         | HTTP Requests
[FRONTEND (Browser)] --------> [BACKEND (Go Server)]
(localhost:3000)               (localhost:3000/api)
         |                              |
     Shows UI                      Stores Data
         |                              |
    index.html                  SQLite Database
    style.css                   (wasatext.db)
    request.js
```

What Each Part Does

- **Frontend:** Shows the user interface (HTML), handles clicks (JavaScript), and sends requests to the backend.
- **Backend:** Receives requests, processes them (e.g., "save this message"), talks to the database, and sends responses.

Part 4: REST APIs — The Rules of Communication

What is an API?

API = Application Programming Interface. It's a **contract** between the frontend and backend. It says: *"If you send me THIS request, I will give you THAT response."*

What Makes it "REST"?

REST (Representational State Transfer) is a style of designing APIs. It has strict rules:

Rule 1: Use URLs to identify "Resources" A "resource" is a thing. In WASAText:

- Users are resources: `/users`

- Conversations: `/conversations`
- Messages: `/conversations/123/messages` (Hierarchy: Message is *inside* Conversation)

Rule 2: Use HTTP Methods to indicate Actions

Method	Meaning	Example
GET	Read/Retrieve	`GET /conversations` (Get all my chats)
POST	Create	`POST /conversations` (Start a new conversation)
PUT	Update/Replace	`PUT /users/5/username` (Change name)
DELETE	Remove	`DELETE /messages/10` (Delete message)

Rule 3: Use Status Codes to indicate Results

Code	Meaning	When to Use
200	OK	Request succeeded
201	Created	Something was created (e.g., new message)
204	No Content	Success, but nothing to return (like DELETE)
400	Bad Request	You sent invalid data (e.g., empty message)
401	Unauthorized	You need to log in
404	Not Found	That thing doesn't exist
500	Server Error	Something broke on the server

Part 5: Authentication — Who Are You?

The Problem

When you send a message, the server needs to know WHO is sending it. In real apps, this involves Passwords, Encryption, and Complex tokens (JWTs).

Our Simplified Solution

For this project, we use a **simplified authentication**:

1. You send your username (`POST /session`).
2. The server checks if you exist. If not, it creates you.
3. The server gives you back a **User ID** (e.g., `550e8400...`).
4. You include that ID in **EVERY** request header: `Authorization: Bearer 550e8400...`

Oral Exam Defense:

> *"I used a simplified Bearer token model where the User ID acts as the token. In a real production app, I would replace this with a JWT (JSON Web Token) which is cryptographically signed and cannot be forged. This design allows us to demonstrate the architecture of authentication headers without the complexity of cryptography."*

Part 6: The Data Model — What We Store

Entity-Relationship Diagram

erDiagram USER ||--o{ CONVERSATION : "belongs to (many-to-many)" USER ||--o{ MESSAGE : "sends" CONVERSATION ||--o{ MESSAGE : "contains" USER { string id PK string username blob photo } CONVERSATION { string id PK boolean is_group string group_id } MESSAGE { string id PK string content string sender_id FK string conversation_id FK datetime timestamp }

Understanding Relationships

- **User <-> Conversation (Many-to-Many):** One user can be in MANY conversations. One conversation can have MANY users.
- **User <-> Message (One-to-Many):** One user can send MANY messages. Each message has exactly ONE sender.
- **Conversation <-> Message (One-to-Many):** One conversation has MANY messages. Each message belongs to ONE conversation.

Part 7: Why SQLite?

What is a Database?

A "database" is where we store data permanently.

Why SQLite?

1. **Serverless:** Unlike MySQL or PostgreSQL, SQLite doesn't need a separate server process running in the background.
2. **File-Based:** The entire database is just a single file (`wasatext.db`) on your disk.
3. **Standard SQL:** It supports full SQL (SELECT, INSERT, JOIN), so the skills transfer to "real" databases.

Oral Exam Defense:

> *"I chose SQLite because it provides a full relational database experience (SQL, Tables, Foreign Keys) without the operational overhead of managing a separate database server. It is perfect for this educational project because it is file-based and easy to back up."*

Summary Checklist

Before moving to Chapter 1, make sure you understand:

- □ The difference between Frontend and Backend
- □ What HTTP Request and Response are
- □ The 4 main HTTP methods (GET, POST, PUT, DELETE)
- □ What REST means and its core principles
- □ How our simplified authentication works
- □ The 3 main entities (User, Conversation, Message)
- □ Why we use a file-based database (SQLite)