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 WhatsApp Web: - The **Frontend** is what you see: the chat bubbles, the typing box, the list of contacts - The **Backend** is what you don't see: the computer storing all your messages

Why Split It?

Imagine you're building a library: - The **Frontend** is the reception desk where people ask for books - The **Backend** is the 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 warehouse 3. **Separation of concerns**: The receptionist doesn't need to know how books are organized

---

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.

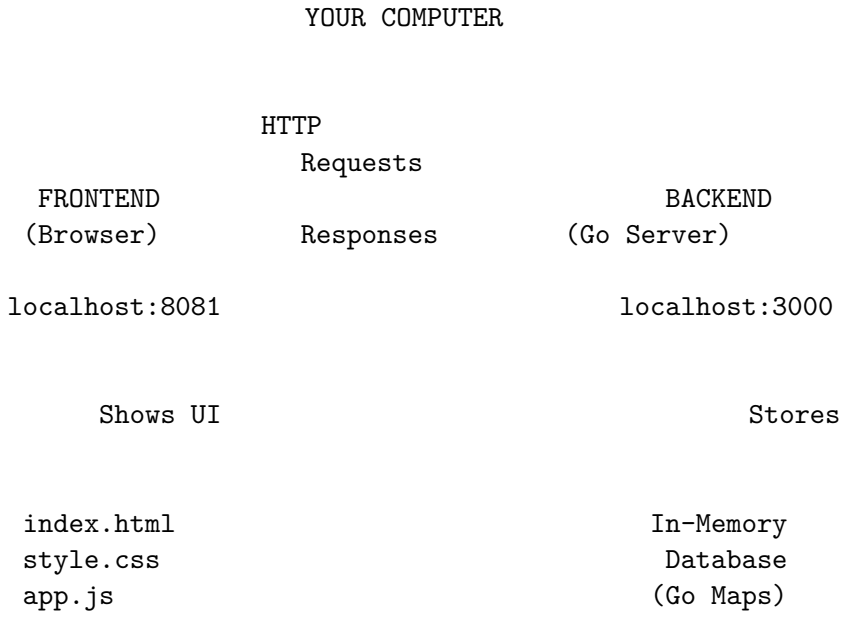
An HTTP Request has: - **Method**: What you want to do (GET, POST, PUT, DELETE) - **URL**: Where you want to do it (`/messages`) - **Headers**: Extra information (like your login token) - **Body**: Data you're sending (like the message text)

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



What Each Part Does

Frontend (Port 8081): - Shows the user interface (HTML) - Handles user clicks and typing (JavaScript) - Sends requests to the backend when user does something - Displays the responses

Backend (Port 3000): - Receives requests from the frontend - Processes them (e.g., "save this message") - Stores data in memory - Sends responses back

---

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 rules:

Rule 1: Use URLs to identify "Resources" A "resource" is a thing. In WASAText: - Users are resources: `/users` - Conversations are resources: `/conversations` - Messages are resources: `/conversations/123/messages`

Notice how messages are "inside" conversations? That's because a message cannot exist without a conversation. This is called **resource hierarchy**.

Rule 2: Use HTTP Methods to indicate Actions

Method	Meaning	Example
GET	Read/Retrieve	GET /conversations Get all my chats
POST	Create	POST /conversations Create a new chat
PUT	Update/Replace	PUT /users/5/name Change user 5's name
DELETE	Remove	DELETE /messages/10 Delete message 10

Rule 3: Use Status Codes to indicate Results

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

Example: Sending a Message

Request:

```
POST /conversations/5/messages
Authorization: Bearer 123
Content-Type: application/json
```

```
{
  "content": "Hello, world!"
}
```

Translation: - Method: POST → "I want to CREATE something" - URL: /conversations/5/messages → "A message in conversation 5" - Authorization: Bearer 123 → "I am user 123" - Body: The actual message content

Response:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "id": 42,
  "content": "Hello, world!",
  "sender_id": 123,
  "timestamp": "2024-01-01T12:00:00Z"
}
```

Translation: - 201 Created → "Success, I made your message" - Body: The message that was created, with extra info the server added

---

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 (stored securely with hashing) - Sessions or JWTs (special encrypted tokens) - Expiration times - Refresh tokens

Our Simplified Solution

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

1. You send your username
2. The server gives you back a number (your user ID)
3. You include that number in EVERY request: `Authorization: Bearer 123`
4. The server trusts that number

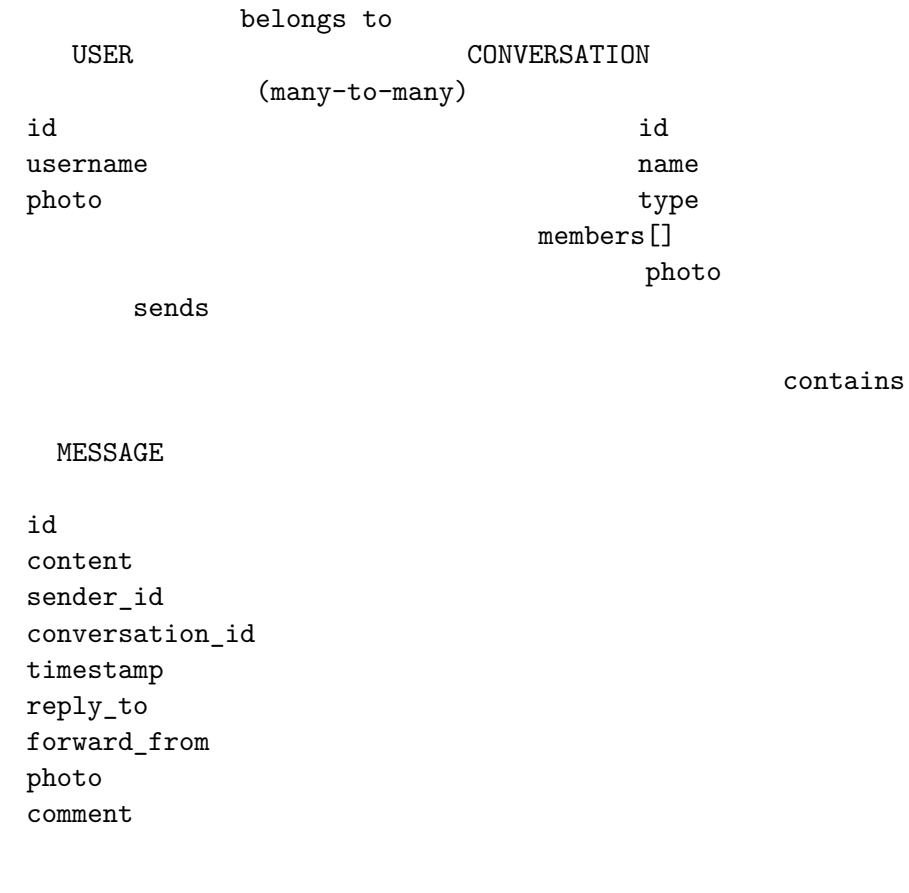
Why is this insecure in real life? Because anyone can guess your number. If I say "I am user 456", the server believes me.

Why is it OK for this project? Because the exam tests your understanding of **architecture**, not security. The professor knows real auth is complex.

Oral Exam Defense: "I used a simplified Bearer token model where the user ID acts as the token. In production, I would replace this with JWT (JSON Web Token) which is cryptographically signed and cannot be forged."

Part 6: The Data Model — What We Store

Entity-Relationship Diagram



Understanding Relationships

User Conversation (Many-to-Many): - One user can be in MANY conversations - One conversation can have MANY users - This is stored as `members[]` array in Conversation

User Message (One-to-Many): - One user can send MANY messages - Each message has exactly ONE sender - Stored as `sender_id` in Message

Conversation Message (One-to-Many): - One conversation has MANY messages - Each message belongs to ONE conversation - Stored as `conversation_id` in Message

---

Part 7: Why In-Memory Database?

What is In-Memory?

A “database” is where we store data permanently. Examples: - PostgreSQL, MySQL, MongoDB (real databases) - Files on disk - **Memory (RAM)** What we use

In-memory means data lives in the computer's RAM. When you restart the program, everything is gone.

Why Use It?

1. **Simplicity**: No setup, no installation, no configuration
2. **Speed**: RAM is 1000x faster than disk
3. **Focus**: The exam tests architecture, not database administration

The Trade-off

If the server crashes, all data is lost. In production, you'd use PostgreSQL or MongoDB.

Oral Exam Defense: "I used an in-memory database with Go maps for simplicity. This demonstrates my understanding of data structures without the complexity of SQL. In production, I would use PostgreSQL for persistence and add a caching layer like Redis for performance."

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 an in-memory database

Glossary

Term	Definition
API	A contract defining how systems communicate
REST	A style of API design using resources and HTTP methods
HTTP	The protocol browsers use to talk to servers
Request	What you send TO the server
Response	What you get BACK from the server
Frontend	The part running in your browser
Backend	The part running on a server
In-Memory	Stored in RAM, lost when program stops
Bearer Token	A string sent with requests to prove identity