□ Chapter 3: The Face — Frontend Development in JavaScript

Introduction: What You Will Learn

The frontend is what users see. Without the backend, it's just a pretty picture. Without the frontend, users can't interact with your app.

By the end of this chapter, you will understand: 1. How HTML, CSS, and JavaScript work together 2. What the DOM is and how to manipulate it 3. How to make HTTP requests with `fetch` 4. What `async/await` means and why we need it 5. How to manage application state 6. How polling simulates real-time updates

---

Part 1: The Three Pillars of Web

HTML: The Structure

HTML (HyperText Markup Language) defines WHAT is on the page.

```html
<div id="chat">
    <h1>Welcome to WASAText</h1>
    <input type="text" id="username-input" placeholder="Enter username">
    <button id="login-btn">Login</button>
</div>
```

Key concepts: - **Tags**: `<div>`, `<h1>`, `<input>` are tags - **Attributes**: `id="chat"`, `type="text"` are attributes - **id**: A unique identifier we use in JavaScript to find elements

CSS: The Style

CSS (Cascading Style Sheets) defines HOW things look.

```css
#chat {
    background-color: #f5f5f5;
    padding: 20px;
    border-radius: 8px;
}

button {
    background-color: #008069;
    color: white;
    padding: 10px 20px;
}
```

Key concepts: - **Selectors**: `#chat` (id), `.class-name` (class), `button` (tag) - **Properties**: `background-color`, `padding`, etc. - **Values**: `#008069`, `20px`, etc.

JavaScript: The Behavior

JavaScript defines WHAT HAPPENS when users interact.

```
document.getElementById("login-btn").addEventListener("click", function() {
    const username = document.getElementById("username-input").value;
    alert("Hello, " + username);
});
```

This says: "When the login button is clicked, get the username and show an alert."

Part 2: Understanding the DOM

What is the DOM?

DOM = Document Object Model

When the browser loads HTML, it creates a tree structure in memory:

```
document
  html
      head
          title
      body
          div#app
              div#login-view
                  form
                      input
                      button
              div#main-view (hidden)
```

JavaScript can: - **Find** elements: `document.getElementById("login-view")` - **Change** elements: `element.innerHTML = "New content"` - **Hide** elements: `element.classList.add("hidden")` - **Create** elements: `document.createElement("div")`

Finding Elements

```
// By ID (returns ONE element)
const loginBtn = document.getElementById("login-btn");

// By class (returns a LIST of elements)
const buttons = document.getElementsByClassName("btn");

// By CSS selector (returns ONE element)
const firstMessage = document.querySelector(".message");

// By CSS selector (returns a LIST)
const allMessages = document.querySelectorAll(".message");
```

Changing Elements

```
const element = document.getElementById("my-element");
```

```
// Change text content
element.textContent = "Hello";  // Plain text

// Change HTML content
element.innerHTML = "<strong>Bold</strong> text";  // Can include tags

// Change styles
element.style.backgroundColor = "red";

// Add/remove CSS classes
element.classList.add("hidden");
element.classList.remove("hidden");
element.classList.toggle("active");
```

Creating Elements

```
// Create a new div
const div = document.createElement("div");

// Set its content
div.className = "message-bubble msg-mine";
div.innerHTML = `
    <div class="msg-content">Hello, world!</div>
    <span class="msg-meta">12:34</span>
`;

// Add it to the page
document.getElementById("messages-list").appendChild(div);
```

Part 3: The Fetch API — Talking to the Backend

What is fetch?

`fetch` is the modern way to make HTTP requests from JavaScript.

Basic GET Request

```
// Get all conversations
fetch("http://localhost:3000/conversations")
    .then(response => response.json())
    .then(data => {
        console.log(data);  // Array of conversations
    })
    .catch(error => {
        console.error("Failed:", error);
    });
```

POST Request with Body

```
fetch("http://localhost:3000/session", {
    method: "POST",
    headers: {
        "Content-Type": "application/json"
    },
    body: JSON.stringify({
        username: "alice"
    })
})
.then(response => response.json())
.then(data => {
    console.log("Logged in as:", data);
});
```

With Authentication

```
fetch("http://localhost:3000/conversations", {
    headers: {
        "Authorization": "Bearer 123"  // Your token
    }
})
.then(response => response.json())
.then(data => console.log(data));
```

Part 4: async/await — Making Async Code Readable

The Problem with Promises

Promises use `.then().then().then()`, which gets messy:

```
fetch("/conversations")
    .then(response => response.json())
    .then(conversations => {
        return fetch("/conversations/" + conversations[0].id + "/messages");
    })
    .then(response => response.json())
    .then(messages => {
        console.log(messages);
    });
```

The Solution: async/await

`async/await` makes asynchronous code look synchronous:

```
async function loadMessages() {
    const response1 = await fetch("/conversations");
    const conversations = await response1.json();

    const response2 = await fetch("/conversations/" + conversations[0].id + "/messages");
    const messages = await response2.json();

    console.log(messages);
}
```

What Does `await` Do?

`await` says: "Pause here until this operation completes, then continue."

Important: You can only use `await` inside an `async` function.

Error Handling

```
async function loadData() {
    try {
        const response = await fetch("/conversations");
        if (!response.ok) {
            throw new Error("HTTP error: " + response.status);
        }
        const data = await response.json();
        return data;
    } catch (error) {
        console.error("Failed to load:", error);
        return [];
    }
}
```

Oral Exam Defense: "JavaScript is single-threaded. Using `await` allows the browser to continue processing user interactions while waiting for the server response. Without it, the UI would freeze during every network request."

Part 5: Our Frontend Code Explained

index.html Structure

```
<div id="app">
    <!-- Login Screen -->
    <div id="login-view" class="view">
        ...
    </div>

    <!-- Main Chat Interface -->
    <div id="main-view" class="view hidden">
        ...
    </div>
</div>
```

Logic: Both views exist in the HTML, but only one is visible. We toggle the `hidden` class.

State Management

```
let state = {
    user: null,              // Current logged-in user
    conversations: [],        // List of all conversations
    currentConversationId: null,  // Which chat is open
    messages: {},            // Cache: convId -> [messages]
    pollingInterval: null    // Timer for auto-refresh
};
```

What is state? State is all the data your app needs to know at any moment: - Who am I? (`user`) - What chats do I have? (`conversations`) - What chat am I looking at? (`currentConversationId`)

Why a global object? - Easy to access from anywhere - Clear what data the app depends on - Can save to localStorage for persistence

The API Wrapper

```
async function apiCall(endpoint, method = "GET", body = null) {
    const headers = {
        "Content-Type": "application/json"
    };

    // Add auth header if logged in
    if (state.user) {
        headers["Authorization"] = `Bearer ${state.user.token}`;
    }

    const config = {
        method,
        headers,
    };

    if (body) {
        config.body = JSON.stringify(body);
    }

    try {
        const response = await fetch(`http://localhost:3000${endpoint}`, config);

        if (response.status === 204) {
            return null;  // No content (for DELETE)
        }
        if (!response.ok) {
            const errData = await response.json().catch(() => ({}));
            throw new Error(errData.error || `Error ${response.status}`);
        }

        return await response.json();
    } catch (err) {
        console.error("API Call Failed:", err);
        throw err;
    }
}
```

Why a wrapper? Instead of writing `fetch(...)` with headers every time, we write it once. Benefits: 1. Automatically adds auth token 2. Automatically parses JSON 3. Centralized error handling 4. Easy to add logging or retry logic later

Login Flow

```
async function doLogin(username) {
    try {
        // 1. Call the backend
        const data = await apiCall("/session", "POST", { username });

        // 2. Save user info in state
        state.user = {
            id: data.id,
            token: data.token,
            username: username
        };

        // 3. Persist to localStorage (survives refresh)
        localStorage.setItem("wasaUser", JSON.stringify(state.user));

        // 4. Update the UI
        showMainView();

        // 5. Start fetching data
        startPolling();
    } catch (err) {
        document.getElementById("login-error").innerText = err.message;
    }
}
```

Rendering the Conversation List

```
function renderConversationList() {
    const list = document.getElementById("conversations-ul");
    list.innerHTML = "";  // Clear existing items
    state.conversations.forEach(conv => {
        // Create list item
        const li = document.createElement("li");
        li.className = `conv-item ${state.currentConversationId === conv.id ? "active" : ""}`;

        // Set content
        let name = conv.name || `Chat ${conv.id}`;
        li.innerHTML = `
            <div class="avatar">${name.charAt(0).toUpperCase()}</div>
            <div class="conv-info">
                <strong>${name}</strong>
                <div style="font-size:0.8rem; color:#888;">${conv.type}</div>
            </div>
        `;

        // Add click handler
        li.onclick = () => selectConversation(conv);

        // Add to list
        list.appendChild(li);
    });
}
```

Step by step: 1. Find the `<ul>` element 2. Clear its contents (`innerHTML = ""`) 3. Loop through conversations 4. For each one, create a `<li>` element 5. Set its HTML content using template literals 6. Add a click handler 7. Append to the list

Rendering Messages

```
function renderMessages(convId) {
    const list = document.getElementById("messages-list");
    const msgs = state.messages[convId] || [];
    list.innerHTML = "";

    msgs.forEach(msg => {
        const div = document.createElement("div");
        const isMine = msg.sender_id === state.user.id;
        div.className = `message-bubble ${isMine ? "msg-mine" : "msg-theirs"}`;

        div.innerHTML = `
            <div class="msg-content">${msg.content}</div>
            <span class="msg-meta">${new Date(msg.timestamp).toLocaleTimeString()}</span>
        `;
        list.appendChild(div);
    });

    // Scroll to bottom
    list.scrollTop = list.scrollHeight;
}
```

Key logic: - `msg.sender_id === state.user.id`: Is this MY message? - If yes, add class `msg-mine` (appears on right, green) - If no, add class `msg-theirs` (appears on left, white) - After rendering, scroll to bottom so newest message is visible

---

Part 6: Polling — Simulating Real-Time

The Problem

How do we know when someone sends us a message? The server can't "push" data to the browser (without WebSockets).

The Solution: Polling

Every few seconds, ask the server: "Anything new?"

```
function startPolling() {
    // Immediately load data
    loadConversations();

    // Then load every 2 seconds
    state.pollingInterval = setInterval(() => {
        loadConversations();

        if (state.currentConversationId) {
            loadMessages(state.currentConversationId);
        }
    }, 2000);  // 2000 milliseconds = 2 seconds
}
```

How setInterval Works

```
setInterval(function, delay);
```

- `function`: What to do
- `delay`: How often (in milliseconds)

Returns an ID that you can use to stop it:

```
const intervalId = setInterval(..., 2000);

// Later, to stop:
clearInterval(intervalId);
```

Trade-offs of Polling

Pros: - Simple to implement - Works everywhere - No special server setup

Cons: - Wastes bandwidth (mostly empty responses) - Delays (up to 2 seconds to see new messages) - Server load (every client asking constantly)

Better alternatives (for real apps): - **WebSockets**: Persistent connection, server pushes updates - **Server-Sent Events (SSE)**: One-way push from server - **Long Polling**: Server holds request until there's data

Oral Exam Defense: "I used polling for simplicity. Every 2 seconds, the client requests updates. In production, I would implement WebSockets for instant updates and reduced server load. The polling interval of 2 seconds is a balance between responsiveness and resource usage."

Part 7: Event Handling

Adding Event Listeners

```
document.getElementById("login-btn").addEventListener("click", function() {
    // Handle click
});

// Arrow function version (shorter)
document.getElementById("login-btn").addEventListener("click", () => {
    // Handle click
});

// With event object
document.getElementById("login-btn").addEventListener("click", (event) => {
    event.preventDefault();  // Prevent default behavior
    // Handle click
});
```

Form Submission

```
document.getElementById("login-form").addEventListener("submit", (e) => {
    e.preventDefault();  // Don't reload the page!
    const username = document.getElementById("username-input").value;
    doLogin(username);
});
```

Why `preventDefault()`? By default, submitting a form reloads the page. We want to handle it in JavaScript instead.

DOMContentLoaded

```
document.addEventListener("DOMContentLoaded", () => {
    // This runs when HTML is fully loaded
    // Put all your setup code here
});
```

Why? If JavaScript runs before HTML loads, `document.getElementById(...)` returns `null` because the element doesn't exist yet.

Part 8: localStorage — Persistence

The Problem

When you refresh the page, all JavaScript variables are reset. You're logged out!

The Solution: localStorage

`localStorage` stores data in the browser permanently (until cleared).

```
// Save data
localStorage.setItem("wasaUser", JSON.stringify(state.user));

// Load data
const saved = localStorage.getItem("wasaUser");
if (saved) {
    state.user = JSON.parse(saved);
}

// Remove data
localStorage.removeItem("wasaUser");
```

In Our App

On login:

```
localStorage.setItem("wasaUser", JSON.stringify(state.user));
```

On page load:

```
const saved = localStorage.getItem("wasaUser");
if (saved) {
    state.user = JSON.parse(saved);
    showMainView();
    startPolling();
}
```

On logout:

```
localStorage.removeItem("wasaUser");
location.reload();  // Refresh the page
```

Summary Checklist

Before moving to Chapter 4, make sure you understand:

- ☐ How HTML, CSS, and JavaScript work together
- ☐ What the DOM is
- ☐ How to find and modify elements
- ☐ How `fetch` makes HTTP requests
- ☐ What `async/await` does
- ☐ How state management works
- ☐ How polling simulates real-time updates
- ☐ How localStorage persists data

Oral Exam Questions

Q: Why Vanilla JavaScript instead of React? "React abstracts DOM manipulation, which is great for productivity but hides how things actually work. Using Vanilla JS demonstrates my understanding of the fundamentals—DOM API, event handling, and state management without framework magic."

Q: Explain your state management approach. "I use a single global object to hold all application state. When data changes, I manually re-render the affected components. This is simpler than Redux but sufficient for this scale. Each render function clears its container and rebuilds from state."

Q: Why is polling inefficient? "Every client makes requests every N seconds regardless of whether there's new data. With 1000 users polling every 2 seconds, that's 500 requests per second, mostly returning unchanged data. WebSockets maintain persistent connections and only send data when something changes."