CASE STUDY 2: Real-Time Chat Application

Scenario:
Build a **real-time chat application** where multiple users can send and receive messages. The system should:

- Handle multiple users with separate chat rooms
- Store chat messages temporarily in memory
- Expose a simple HTTP API to send and fetch messages
- Demonstrate **event loop, async operations, modules, and HTTP server**

Step-by-Step Implementation

Step 1: Project Structure

```
chat-app/
  ├─ modules/
  │    ├─ user.js         // User management
  │    ├─ chatRoom.js     // Chat room and message storage
  │    ├─ config.js       // Port and settings
  │    └─ index.js        // Re-export all modules
  └─ server.js            // HTTP server
```

Step 2: User Module

- Create a `User` class with `name`, `email`, and `id`.
- Maintain a global array of all users.
- Add methods to register a user and validate user input.

Concepts: Classes, arrays, module exports.

Step 3: Chat Room Module

- Create a `ChatRoom` class with:
    - `name` of chat room
    - Array of messages (each message has sender, content, timestamp)
- Methods to:

    - Add a message
    - Fetch last N messages
- Messages stored in memory for simplicity.

Concepts: Classes, arrays, timestamps, async-safe operations.

Step 4: Config Module

- Store constants like `PORT` and `MAX_MESSAGE_LENGTH`.
- Export values to use in server.

Step 5: Index Module

- Re-export `User`, `ChatRoom`, and configuration constants.
- Simplifies imports in server.

Step 6: HTTP Server

- Import modules from index.
- Create sample users and chat rooms.
- HTTP endpoints:
    - `/sendMessage` → accepts POST with user & message → adds to chat room
    - `/getMessages` → accepts GET → returns last N messages
- Use async callbacks to handle requests without blocking.

Concepts: HTTP server, routing, async handling, JSON response.

Step 7: Event Loop and Non-Blocking I/O

- Sending and fetching messages handled asynchronously.
- Multiple users can interact without blocking server.

Step 8: Debugging

- Use `console.log` to monitor user registration, messages, and endpoints.
- Optional: `node --inspect server.js` for step debugging.

Outcome

- Real-time chat app with **user management**, **chat rooms**, and **async message handling**.
- Demonstrates **modules, classes, HTTP server, and event loop**.