Template – Sample Course Project Report: Real-Time Chat Application

Contents

---

1. Cover Page & Info

- **Project Title:** Real-Time Chat Application using Event-Driven Architecture (EDA)
- **Course Name:** Software Architecture
- **Student Info:** [Your Name], [Student ID]
- **Date:** December 5, 2025

---

2. Executive Summary

This project's goal was to design and implement a lightweight, multi-user **Real-Time Chat Application**. The core architectural pattern utilized was the **Event-Driven Architecture (EDA)**, specifically implemented using **WebSockets** for bi-directional, persistent communication. The final achievement is a functional application that provides near-instantaneous message broadcasting across multiple client sessions, successfully meeting the core quality attribute of **Real-Time Response**.

---

3. Project Requirements & Goals

This section defines the scope and the key architectural drivers for the project.

3.1 Core Functional Requirements

ID	Description
FR-01	The application must allow users to connect to the chat server using a unique username.
FR-02	The application must allow a connected user to send a text message to the server.
FR-03	The server must broadcast any received message to all other connected users in real-time.
FR-04	The application must display the messages chronologically, including the sender's username.

3.2 Key Quality Attributes (Architectural Goals)

- **Real-Time Response:** The most critical goal. Messages must be delivered and displayed on all connected clients with minimal perceived latency (sub-200ms).
- **Availability:** The server should maintain a persistent connection, minimizing downtime or dropped connections.
- **Modifiability:** The architecture should allow for easy extension, such as adding private messaging or multiple chat rooms, without overhauling the core broadcast logic.

3.3 Use Case Modeling

Use Case	Actor	Flow Description
Join Chat	User	1. User opens the application. 2. User enters a unique username. 3. User establishes a WebSocket connection with the server.
Send Message	User	1. User types a message. 2. User presses Send. 3. Client sends the message and username via the persistent WebSocket.
Receive Message	All Users	1. Server receives a message from User A. 2. Server broadcasts the message to all User A's **io.sockets**. 3. Client displays the new message instantly.

---

4. Architectural Design & Implementation

4.1 Architectural Pattern: Event-Driven Architecture (EDA) via WebSockets

The core of the application is an **Event-Driven Architecture (EDA)**. This pattern is ideal because the server acts primarily as a message broker, reacting to client-initiated events ('send message') by generating and pushing new events ('chat message') to all other connected clients. This allows for the required bi-directional, low-latency, and persistent communication.

System Context Diagram

The C4 Model (Level 1: System Context) shows the boundary of the Chat Application system.

The **User/Client (Browser)** interacts with the system via the persistent **WebSocket** connection. There are no other external systems (like a database or external API) required for the core functionality, keeping the design simple and focused on real-time messaging.

4.2 Technical Stack and Data Model

- **Server Framework:** Node.js (High performance for I/O and non-blocking operations).
- **Real-Time Library: Socket.IO** (Provides a reliable, event-based abstraction over WebSockets).
- **Client:** Vanilla HTML/CSS/JavaScript.
- **Data Model:** The system has no persistent data model (e.g., database) as messages are not stored long-term. The active data model is the **in-memory map of active sockets** managed by Socket.IO.

4.3 Implementation: Server-Side Logic (Node.js/Socket.IO)

The server logic manages two key events: connection and chat message.

- **Connection Handling:**

```
JavaScript
// Code (Server): Node.js/Socket.IO logic for handling the connection
io.on('connection', (socket) => {
  console.log('A user connected:', socket.id);

  // Listen for a disconnect event
  socket.on('disconnect', () => {
    console.log('User disconnected:', socket.id);
  });

  // Listen for incoming chat messages
  socket.on('chat message', (msg) => {
    // io.emit() broadcasts the message to ALL connected sockets
    // including the sender's own, but typically client logic prevents
    // displaying one's own message twice.
    // The required broadcast logic:
    // Code (Server): Node.js/Socket.IO logic for io.emit('chat message', ...) broadcast.
    io.emit('chat message', msg);
  });
});
```

- **Flow:** When a client connects, the io.on('connection', ...) block executes, registering listeners for that specific socket. When the chat message event is received, the message is immediately broadcast using io.emit() to all clients, demonstrating the core event-driven communication.

4.4 Implementation: Client-Side Logic (JavaScript)

The client-side logic is responsible for establishing the WebSocket connection and, crucially, listening for the broadcasted event from the server to update the user interface.

- **Listener and DOM Update:**

  JavaScript

  // Code (Client): JavaScript snippet showing the socket.on('chat message', ...) listener updating the DOM.

  var socket = io();

  // Listen for incoming messages from the server

  socket.on('chat message', function(msg) {

    // Create a new list item for the message

    var item = document.createElement('li');

    item.textContent = msg.username + ": " + msg.message;

    // Append it to the message list in the DOM

    $('#messages').append(item);

    // Scroll to the bottom

    window.scrollTo(0, document.body.scrollHeight);

  });

- **Flow:** The socket.on('chat message', ...) function is the **event listener**. When the server executes the broadcast (io.emit), this client-side function immediately triggers, allowing the new message to appear in the DOM without any page refresh or polling.

---

5. Testing & Verification

The primary verification for this project is to prove the **real-time, synchronous broadcast** capability.

Verification Proof: Real-Time Broadcast

Test Scenario: Message propagation across two distinct client sessions.

Step	Expected Result	Actual Result	Status
1. Open Browser Tab A (User: Alice)	Connection established.	OK	Pass
2. Open Browser Tab B (User: Bob)	Connection established.	OK	Pass
3. Alice sends message: "Hello Bob!"	Bob's screen instantly updates with "Alice: Hello Bob!"	Instantaneous update on both tabs.	Pass

- **Verification:** Screenshot showing two separate browser tabs with two different usernames, demonstrating that a message sent in one tab appears instantly in the other tab.

---

6. Conclusion & Reflection

This project successfully implemented a real-time chat application using an Event-Driven Architecture facilitated by Node.js and Socket.IO. The core architectural goal of **Real-Time Response** was met, with verification confirming instantaneous message propagation.

6.1 Lessons Learned

- **EDA Efficiency:** The use of WebSockets and the EDA model proved far more efficient for real-time applications than traditional request-response (REST) or polling models, eliminating unnecessary latency.
- **Broadcast Simplicity:** Socket.IO handles the complex task of socket management and reliable broadcasting, allowing the focus to remain on the application logic.

6.2 Future Improvements

1. **Adding Security:** Implement user authentication (e.g., JWT) to secure the initial WebSocket handshake.

2. **Deployment:** Deploy the server using a platform like AWS ECS or Kubernetes and integrate a dedicated message broker (like RabbitMQ or Redis Pub/Sub) for horizontal scaling beyond a single Node.js instance.
3. **Persistence:** Integrate a database (e.g., MongoDB) to store message history, allowing new users to see past conversations.