Web Chat Application

System Design Document

Rohan Gupta
Indian Institute of Technology Jodhpur
IBY Project

Contents

Introduction

Purpose

This document outlines the system architecture and design of the Web Chat Application. It provides a comprehensive understanding of the system, including its components, interactions, scalability, security considerations, and the rationale behind the technology choices.

Scope

The Web Chat Application is a real-time messaging platform built using the MERN stack. The system is designed for scalability, high availability, and efficient real-time communication for users. The document covers design patterns, deployment strategies, security, fault tolerance, and non-functional requirements.

Definitions, Acronyms, and Abbreviations

- **MERN:** MongoDB, Express.js, React.js, Node.js
- **API:** Application Programming Interface
- **JWT:** JSON Web Token
- **WebSocket:** Full-duplex communication channels over a single TCP connection

High-Level Architecture

System Overview

The Web Chat Application is structured into a front-end client (React.js) and a back-end server (Node.js and Express.js), communicating over WebSockets (Socket.io) for real-time updates. MongoDB is used to store persistent data, such as user information and chat history.

Architectural Diagram

Component Breakdown

- **Front-end (React.js)**: Provides an intuitive UI, manages user state, and interacts with the back-end via WebSockets and HTTP.
- **Back-end (Node.js & Express.js)**: Processes user requests, manages business logic, and handles WebSocket communication.
- **WebSocket (Socket.io)**: Establishes real-time, full-duplex communication between the client and server for instant messaging.
- **Database (MongoDB)**: Stores user profiles, authentication data, and message history.

System Interactions and Workflows

Authentication Flow

1. The user submits login credentials.
2. The front-end sends the credentials via HTTP POST to `/api/auth/login`.
3. The back-end verifies the credentials, generates a JWT, and sends it to the front-end.
4. The JWT is stored in the client's local storage and is included in all subsequent API calls.

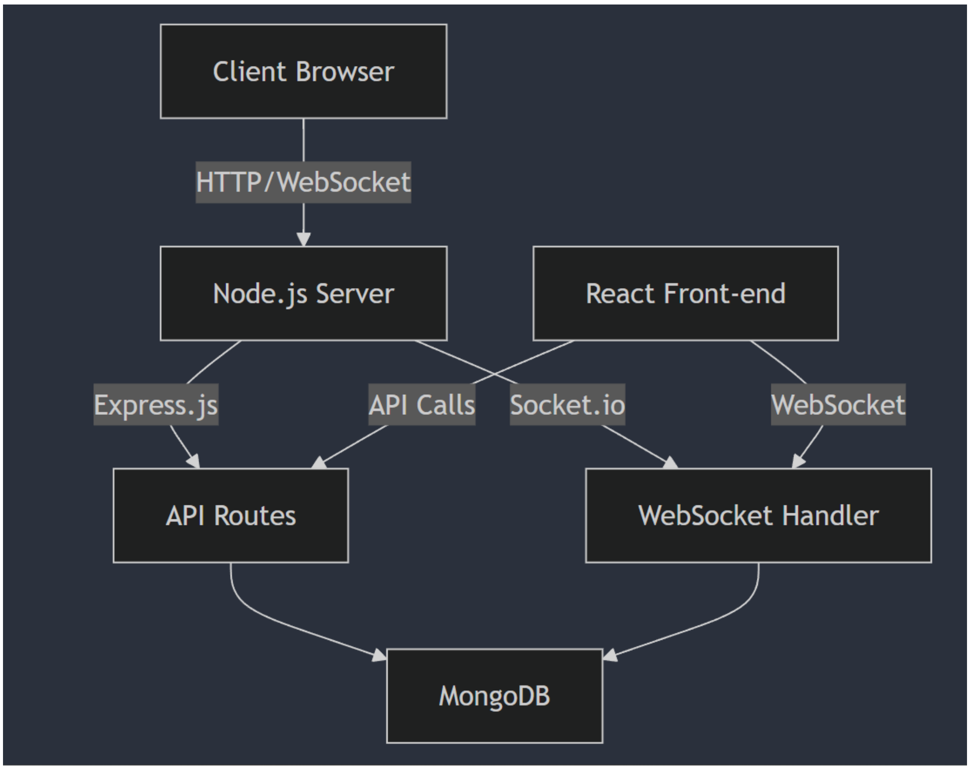

Figure 1: High-Level System Architecture

Messaging Flow

1. User sends a message from the client interface.
2. The message is transmitted to the server via a WebSocket connection.
3. The server validates the message and stores it in MongoDB.
4. The server broadcasts the message to all connected clients within the same chat room.

Sequence Diagram: Message Flow

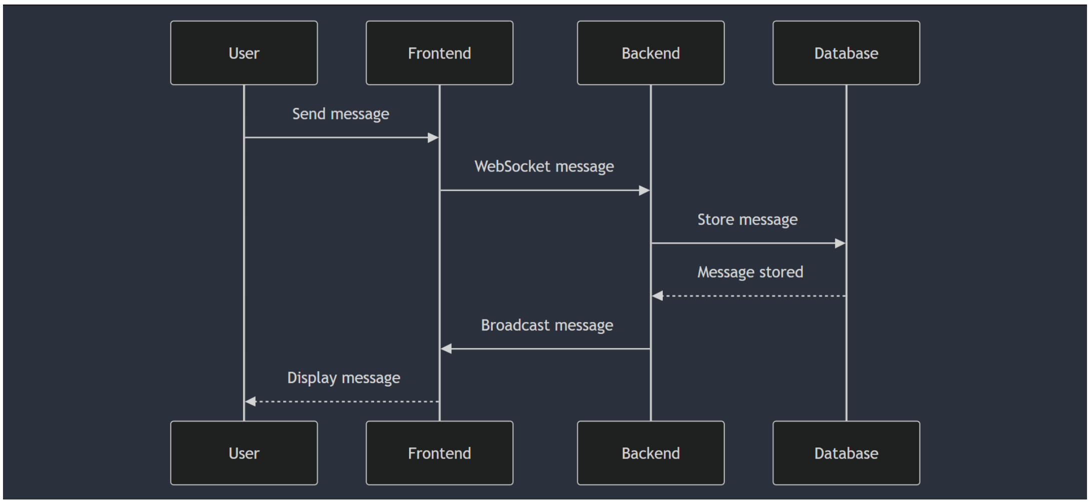

Figure 2: Sequence Diagram of the Messaging Flow

Data Design

Data Description

The system manages two primary data types:

- **User Data:** User profiles and authentication information.
- **Message Data:** Message content, timestamps, sender/receiver information.

Data Dictionary

Entity	Attributes	Data Type	Description
User	id	ObjectId	Unique identifier
	username	String	User's chosen name
	email	String	User's email address
	password	String	Hashed password
Message	id	ObjectId	Unique identifier
	content	String	Message text
	sender	ObjectId	Reference to User
	timestamp	Date	Time of sending

Table 1: Data Dictionary

Scaling and Availability

Scalability Considerations

- **Horizontal Scaling**: Both the front-end and back-end servers can be scaled horizontally by deploying multiple instances behind a load balancer.
- **WebSocket Scaling**: Socket.io can be scaled using Redis Pub/Sub for handling multiple WebSocket server instances.
- **Database Scaling**: MongoDB can be sharded across multiple nodes to support a large user base and chat history.

High Availability

- **Load Balancing**: A load balancer will distribute incoming requests across multiple back-end instances to avoid overloading a single instance.
- **Database Replication**: MongoDB replication will ensure data redundancy and failover support in case a node fails.
- **Session Persistence**: User sessions and WebSocket connections will be persisted to avoid disruptions during scaling or node failures.

Non-Functional Requirements

Performance Metrics

- **Response Time**: Messages should appear in the chat interface within 200ms of being sent.
- **Throughput**: The system should handle up to 1000 concurrent users with minimal latency.
- **Latency**: Real-time WebSocket connections should maintain a latency of less than 100ms for chat operations.

Reliability

The system will ensure 99.9% uptime through load balancing, database replication, and fault tolerance mechanisms.

Security Considerations

- **Authentication**: JWTs are used for secure session management. Tokens are stored client-side and are passed with every request in the Authorization header.
- **Password Hashing**: User passwords are hashed using bcrypt before storage.
- **Transport Layer Security**: All communication between the client and server will be secured using HTTPS and encrypted WebSocket connections.
- **Data Encryption**: Sensitive user information, including messages, will be encrypted at rest in the MongoDB database.

Fault Tolerance

- **WebSocket Reconnection**: The client will automatically attempt to reconnect if the WebSocket connection is interrupted.
- **Server Failover**: In case of server failure, the load balancer will route requests to another instance.
- **Database Failover**: MongoDB will have a replica set configuration to failover seamlessly if a primary node goes down.

Testing

Testing Strategy

- **Unit Testing**: Individual components will be tested using frameworks like Jest for the front-end and Mocha/Chai for the back-end.
- **Integration Testing**: End-to-end tests will be conducted to verify the interactions between front-end and back-end components.
- **User Acceptance Testing (UAT)**: Real users will test the application to validate its functionality and usability before final deployment.

Testing Status

The application has been tested on localhost and all functionalities, including authentication and messaging, are working as intended. However, it has not yet been deployed or tested in a production environment.

Deployment

Deployment Strategy

The deployment of the Web Chat Application will involve the following steps:

- **Environment Setup**: Configure a production environment on cloud services like AWS, Heroku, or DigitalOcean.
- **Containerization**: Use Docker to containerize the application for consistent deployment across environments.
- **Continuous Integration/Continuous Deployment (CI/CD)**: Implement CI/CD pipelines using tools like GitHub Actions or Jenkins for automated testing and deployment.

Monitoring and Logging

Although monitoring tools have not yet been implemented, it is recommended to integrate solutions such as:

- **Application Performance Monitoring (APM)**: Tools like New Relic or Datadog to monitor application performance and detect bottlenecks.
- **Error Tracking**: Implement error tracking tools like Sentry for real-time error reporting.
- **Logging**: Use structured logging (e.g., Winston or Morgan) to capture logs for debugging and monitoring purposes.

Conclusion

The Web Chat Application is designed to provide a robust and scalable real-time messaging solution. This document serves as a guideline for the architecture, design, and future development of the system. Continuous testing, monitoring, and iteration will ensure the application remains efficient and user-friendly.