Advanced DevOps Practices for Enterprise-Scale

Micro SaaS Development

Course	System Administration & Maintenance
Code	IT31023
Term	Intake 11 Term 1
Student Id	ITBNM-2211-0194
Student Name	W.A.D.N Weerawardhana

Contents

Chat Application Documentation

Overview

This documentation provides a comprehensive guide to the real-time chat application. The application allows users to:

- Register with a username
- Send messages in real-time
- See who's currently online
- View typing indicators
- Edit and delete messages

The application is built using Socket.IO for real-time communication, with a Node.js backend and a vanilla JavaScript frontend.

System Architecture

The chat application follows a client-server architecture with these main components:

Client Side

- HTML/CSS UI components
- Client-side JavaScript
- Socket.IO client for real-time communication

Server Side

- Node.js server
- Socket.IO server for handling real-time events
- Database storage for messages and users

Deployment Environments

- Development: Docker Desktop (localhost)
- Production: Railway cloud platform

Communication Flow

The application uses Socket.IO for bidirectional, event-based communication between the client and server:

1. Client establishes WebSocket connection to server
2. Users register with a username
3. Messages are sent/received via Socket.IO events
4. User presence and typing indicators are communicated in real-time
5. Message edits and deletions are synchronized across clients

Key Components

Frontend Components

User Interface Elements

- Sidebar: Displays online users and connection status
- Message Container: Shows chat messages
- Typing Indicator: Shows when users are typing
- Input Area: Allows users to compose messages

JavaScript Functions

- Message Display: Renders messages with appropriate styling
- User Registration: Handles username storage and user authentication
- Typing Detection: Detects when users are typing and broadcasts status
- Edit/Delete Functionality: Allows users to modify or remove their messages

Backend Services

Socket Event Handlers

- User Registration: Handles new user connections
- Message Service: Processes, stores, and broadcasts messages
- Typing Notification Service: Manages typing status updates
- Edit/Delete Service: Handles message modifications

Database Operations

- Storage and retrieval of messages
- User management and status tracking

Socket.IO Events

Client-Emitted Events

- register: Sent when a user provides their username
- send message: Sent when a user sends a chat message
- typing: Sent when a user starts typing
- stop typing: Sent when a user stops typing
- edit message: Sent when a user edits their message
- delete message: Sent when a user deletes their message

Server-Emitted Events

- user list: Updates the list of connected users
- message history: Sends previous messages when a user connects
- new message: Broadcasts a new message to all clients
- user typing: Notifies clients when a user is typing
- user stopped typing: Notifies clients when a user stops typing
- message edited: Broadcasts when a message has been edited
- message deleted: Broadcasts when a message has been deleted
- system message: Sends system notifications to clients

User Flow

1. User Registration:
    - User enters the application and is prompted for a username
    - Username is stored in local Storage for persistent sessions
    - Server receives registration and updates the user list
2. Chat Interaction:
    - User views previous messages (message history)
    - User can see who's online in the sidebar
    - User can compose and send messages
    - Typing indicators show when others are typing
3. Message Management:
    - Users can edit their own messages
    - Users can delete their own messages
    - Edits and deletions are synchronized across all clients

Code Structure

HTML Structure (index.html)

```
<div class="app-container">
  <div class="sidebar">
    <!-- User list and connection status -->
  </div>
  <div class="chat-container">
    <div id="message-container">
      <!-- Messages appear here -->
    </div>
    <div class="input-area">
      <!-- Message input form -->
    </div>
  </div>
</div>
```

Client-Side JavaScript (script.js)

The client-side code handles:

- DOM manipulation for UI updates
- Socket.IO event handling
- User input processing
- Message rendering and management

Key functions include:

- registerUser(): Handles user registration
- addChatMessage(): Renders message elements
- updateUserList(): Updates the online users sidebar
- handleEditMessage(): Manages message editing
- handleDeleteMessage(): Manages message deletion

Server-Side Logic (inferred)

The server handles:

- Socket connections and disconnections
- Message broadcasting
- User status tracking
- Data persistence

Setup and Deployment

Local Development

- Docker Desktop provides containerized environment
- Application runs on localhost with local database

Production Deployment

- Railway platform hosts the application
- API URL: https://chat-app-production-7ff3.up.railway.app
- Persistent database service for message storage

Security Considerations

- Messages are not encrypted end-to-end
- User authentication is minimal (username only)

Performance Optimizations

- Typing indicators use debounce technique (2-second timeout)
- Socket connections handle reconnection automatically
- Messages are loaded from history to avoid data loss

Future Enhancements

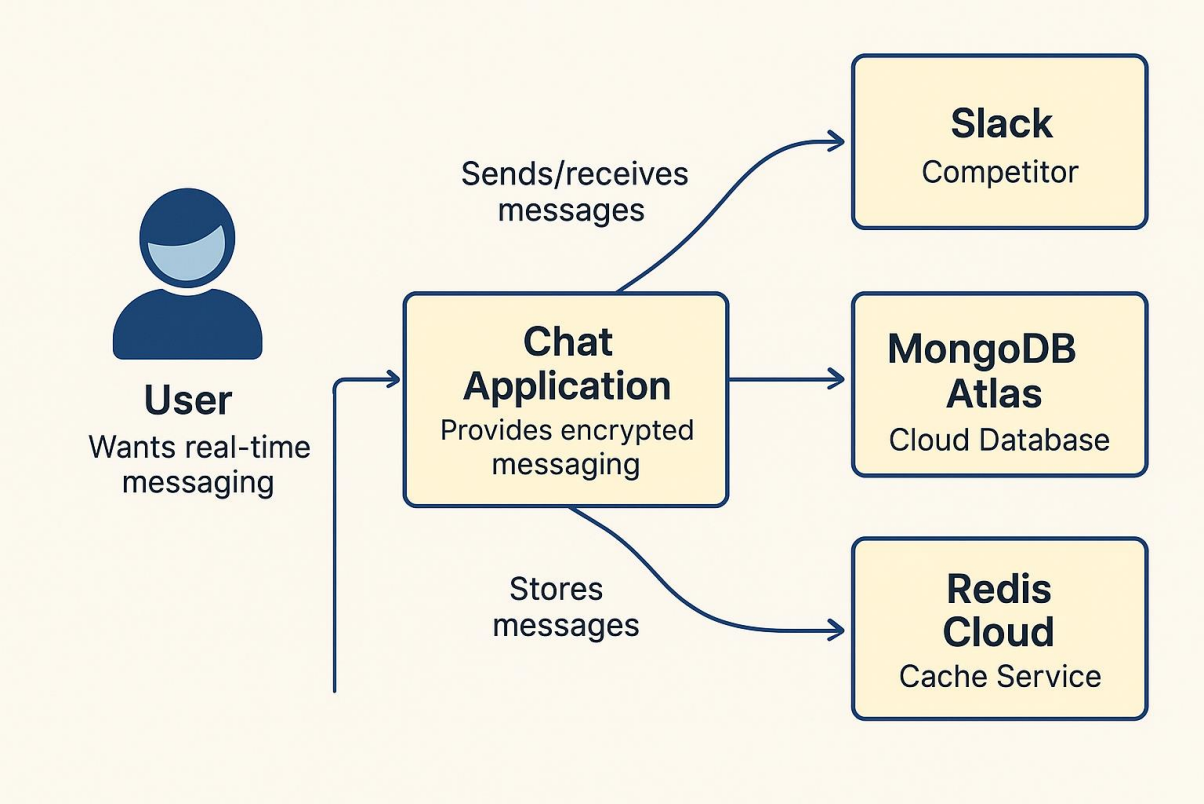
Potential improvements for the application could include:

- User authentication with passwords
- Message encryption
- File/image sharing capabilities
- Private messaging functionality
- Read receipts
- User profiles and avatars

GitHub Repository: Click here

System Architecture (C4 Model)

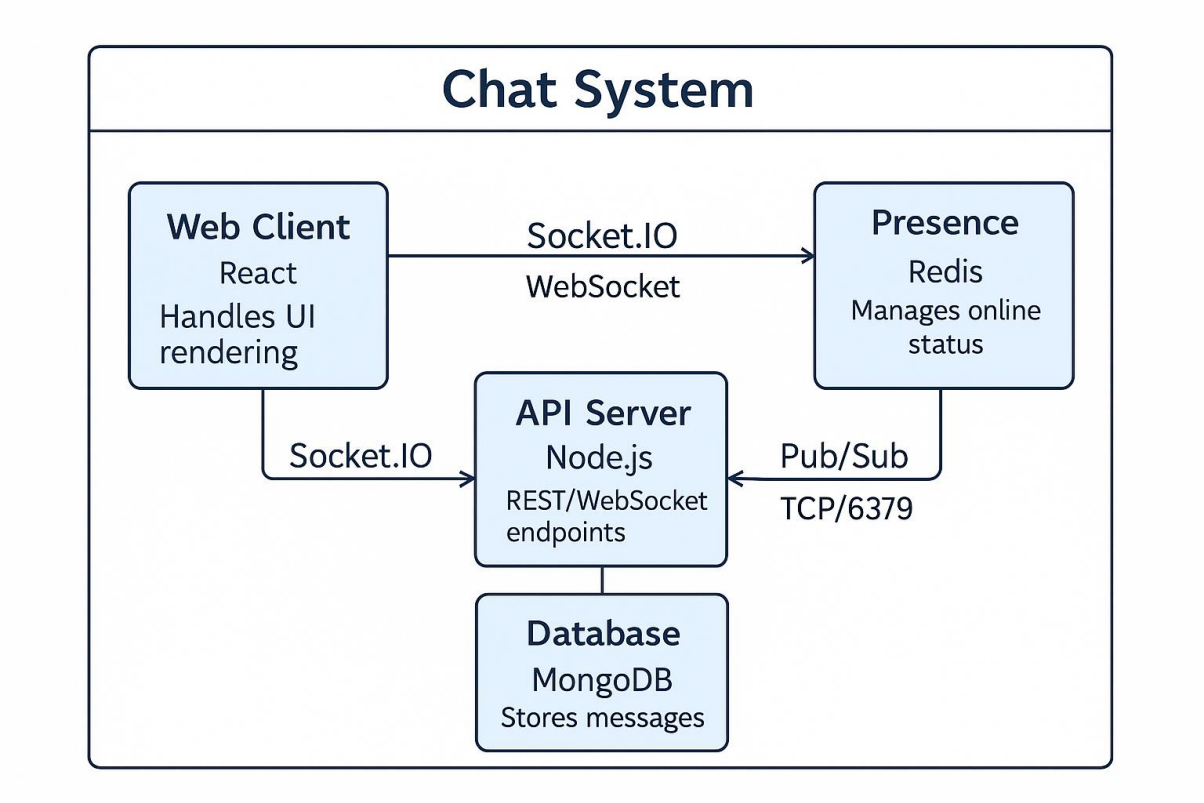
Context Diagram



Explanation:

- Users interact via web/mobile clients
- System provides encrypted messaging

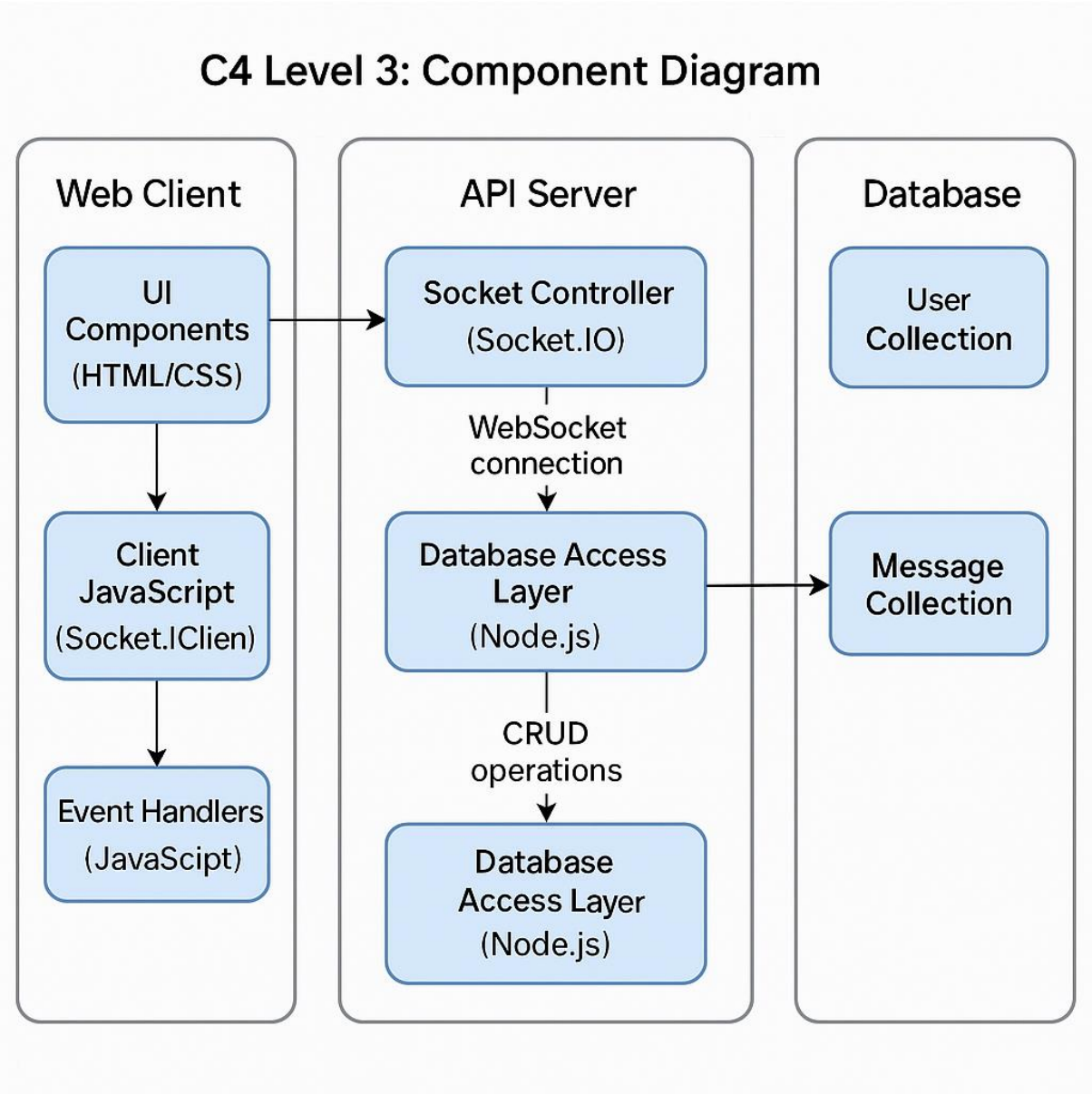
Container Diagram



Key Components:

- **Frontend**: React with Socket.IO client
- **Backend**: Node.js event-driven architecture
- **Data**: MongoDB for messages, Redis for real-time features

Component Diagram



Client-Side Implementation

Socket Connection

The application establishes a Socket.IO connection when the page loads:

```
const socket = io();
```

For production, the application uses a specific API URL:

```
const API_URL = "https://chat-app-production-7ff3.up.railway.app";

const socket = io();
```

User Registration Flow

1. Check for stored username in localStorage
2. If none exists, prompt user for username
3. Register user with server

```
const storedUsername = localStorage.getItem('chatUsername');

const username = storedUsername || prompt('Enter your name:') ||
`User${Math.floor(Math.random() * 1000)}`;


function registerUser(name) {
  socket.emit('register', name, (response) => {
    if (response.status === 'success') {
      localStorage.setItem('chatUsername', name);
      console.log('Registration successful');
    } else {
      alert(`Registration failed: ${response.message}`);
    }
  });
}
```

Message Handling

Messages are sent through the form submission event:

```
form.addEventListener('submit', (e) => {
  e.preventDefault();
  if (input.value.trim()) {
    socket.emit('send message', input.value, (response) => {
      if (response.status === 'error') {
        console.error('Message send failed:', response.message);
        addSystemMessage(`Failed to send message: ${response.message}`);
      }
    });
    input.value = '';
  }
});
```

New messages are received and displayed:

```
socket.on('new message', (data) => {
  const messageElement = addChatMessage(data, data.user === username);
  messageElement.dataset.messageId = data._id;
});
```

Message Display Function

The addChatMessage function creates DOM elements for each message:

```
function addChatMessage(data, isSent = false) {
  const messageElement = document.createElement('div');
  messageElement.className = `message ${isSent ? 'sent' : 'received'}`;
  messageElement.dataset.messageId = data._id || data.id;

  const messageControls = isSent ? `
    <div class="message-controls">
```

```
    <button class="edit-message-btn">Edit</button>
    <button class="delete-message-btn">Delete</button>
  </div>
 ` : '';

 messageElement.innerHTML = `
  <div class="message-info">
   ${!isSent ? `<strong>${data.user}</strong>` : ''}
   <span>${new Date(data.time).toLocaleTimeString()}</span>
  </div>
  <div class="message-text">${data.text}</div>
  ${messageControls}
 `;

 messages.appendChild(messageElement);
 messages.scrollTop = messages.scrollHeight;

 // Add event listeners for edit and delete buttons
 if (isSent) {
  const editBtn = messageElement.querySelector('.edit-message-btn');
  const deleteBtn = messageElement.querySelector('.delete-message-btn');

  editBtn.addEventListener('click', () => handleEditMessage(data));
  deleteBtn.addEventListener('click', () => handleDeleteMessage(data));
 }
 return messageElement;
}
```

Typing Indicator Implementation

Typing detection is implemented with a timeout:

```
let typingTimeout;

input.addEventListener('input', () => {
  socket.emit('typing');
  clearTimeout(typingTimeout);
  typingTimeout = setTimeout(() => {
    socket.emit('stop typing');
  }, 2000);
});

socket.on('user typing', (typingUser) => {
  if (typingUser && typingUser !== username) {
    typingIndicator.textContent = `${typingUser} is typing...`;
    typingIndicator.style.display = 'block';
  } else {
    typingIndicator.textContent = '';
    typingIndicator.style.display = 'none';
  }
});

socket.on('user stopped typing', () => {
  typingIndicator.style.display = 'none';
});
```

Edit and Delete Functionality

Messages can be edited by the original sender:

```
function handleEditMessage(message) {
```

```
  const newText = prompt('Edit your message:', message.text);

  if (newText !== null && newText.trim() !== message.text.trim()) {
    socket.emit('edit message', {
      messageId: message._id || message.id,
      newText: newText.trim()
    }, (response) => {
      if (response.status === 'error') {
        addSystemMessage(`Failed to edit message: ${response.message}`);
      }
    });
  }
}
```

Messages can also be deleted:

```
function handleDeleteMessage(message) {
  if (confirm('Are you sure you want to delete this message?')) {
    socket.emit('delete message', {
      messageId: message._id || message.id
    }, (response) => {
      if (response.status === 'error') {
        addSystemMessage(`Failed to delete message: ${response.message}`);
      }
    });
  }
}
```

Server-Side Implementation (Inferred)

the server-side implementation likely includes:

Socket.IO Event Handlers

```
// User registration
io.on('connection', (socket) => {
  socket.on('register', (username, callback) => {
    // Store username and associate with socket
    users[socket.id] = username;
    // Send success response
    callback({ status: 'success' });
    // Update user list for all clients
    io.emit('user list', Object.values(users));
    // Send message history to new user
    socket.emit('message history', messageHistory);
  });

  // Message handling
  socket.on('send message', (text, callback) => {
    const username = users[socket.id];
    if (!username) {
      return callback({ status: 'error', message: 'Not registered' });
    }

    const message = {
      _id: generateId(),
      user: username,
      text: text,
      time: Date.now()
    };
```

```
  // Store message
  messageHistory.push(message);

  // Broadcast to all clients
  io.emit('new message', message);
  callback({ status: 'success' });
});

// Handle typing events
socket.on('typing', () => {
  const username = users[socket.id];
  socket.broadcast.emit('user typing', username);
});

socket.on('stop typing', () => {
  socket.broadcast.emit('user stopped typing');
});

// Handle edit message
socket.on('edit message', (data, callback) => {
  const { messageId, newText } = data;
  const username = users[socket.id];
```

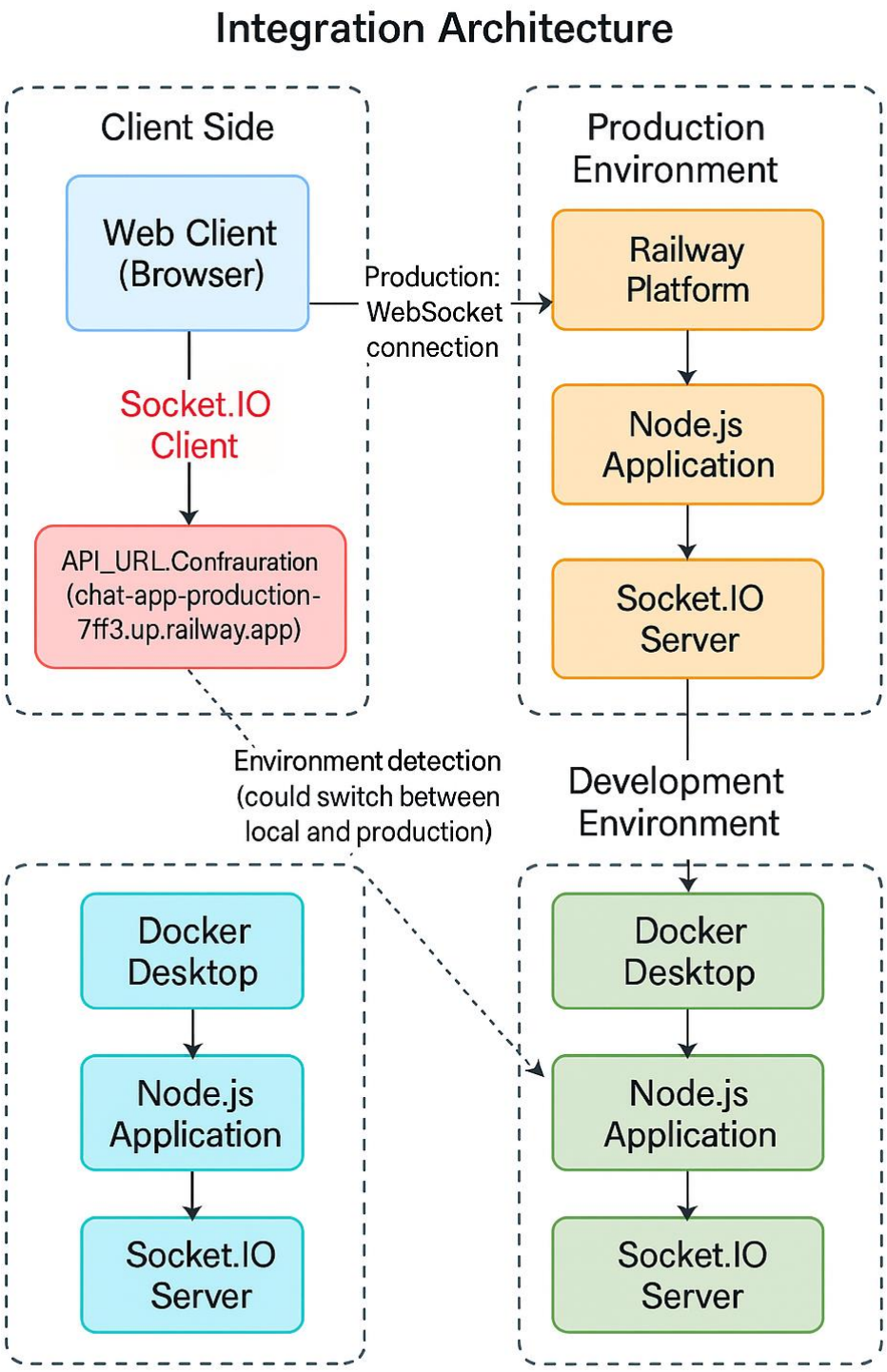
```
  // Find message in storage
  const message = messageHistory.find(msg => msg._id === messageId);

  if (!message || message.user !== username) {
    return callback({ status: 'error', message: 'Cannot edit this message' });
  }

  // Update message
  message.text = newText;

  // Broadcast edit to all clients
  io.emit('message edited', { id: messageId, newText });
  callback({ status: 'success' });
});

// Handle delete message
socket.on('delete message', (data, callback) => {
  const { messageId } = data;
  const username = users[socket.id];

  // Find message index
  const index = messageHistory.findIndex(msg => msg._id === messageId);

  if (index === -1 || messageHistory[index].user !== username) {
    return callback({ status: 'error', message: 'Cannot delete this message' });
  }
```

```
    // Remove message
    messageHistory.splice(index, 1);

    // Broadcast deletion to all clients
    io.emit('message deleted', messageId);
    callback({ status: 'success' });
  });

  // Handle disconnection
  socket.on('disconnect', () => {
    const username = users[socket.id];
    if (username) {
      delete users[socket.id];
      io.emit('user list', Object.values(users));
      io.emit('system message', `${username} has left the chat`);
    }
  });
});
```

Database Integration (Inferred)

The server likely uses a database (MongoDB based on the structure) to store:

1. Messages
2. User information



Database operations would include:

- Inserting new messages
- Updating edited messages
- Deleting messages
- Tracking user presence

Deployment Architecture

Docker Setup (Development)

A typical docker-compose.yml for this application might include:

```
version: '3'

services:

  app:

    build: .

    ports:

      - "3000:3000"

    environment:

      - NODE_ENV=development

      - MONGODB_URI=mongodb://mongo:27017/chatapp

    depends_on:

      - mongo

    volumes:

      - ./:/app

      - /app/node_modules


  mongo:

    image: mongo:latest
```

```
    ports:
      - "27017:27017"
    volumes:
      - mongo_data:/data/db

volumes:
  mongo_data:
```

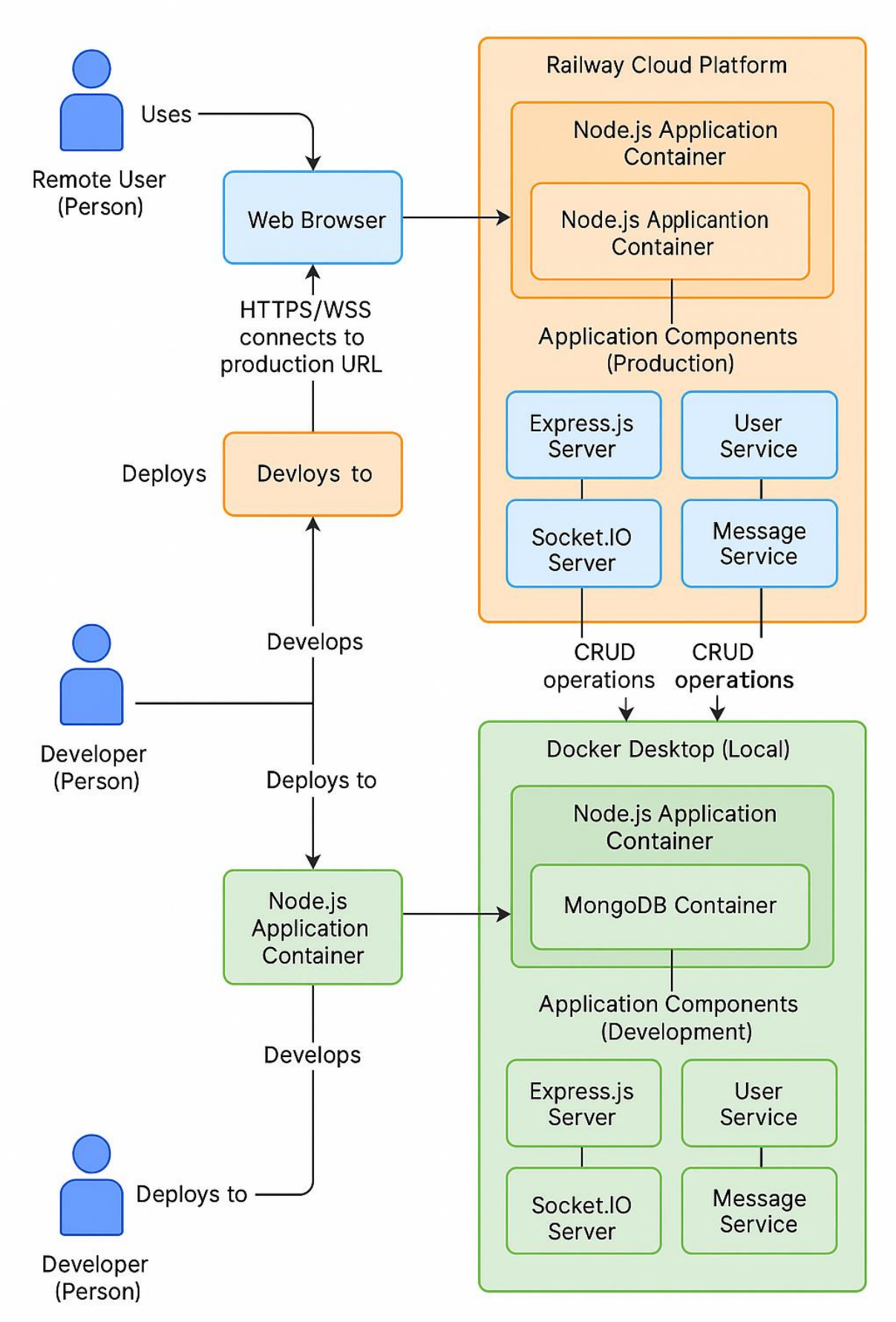
Railway Deployment (Production)

The application is deployed to Railway with:

- Node.js service running the application
- MongoDB service for data persistence
- Environment variables for configuration

Performance Considerations

1. Connection Management
    - Socket.IO handles reconnection automatically
    - Connection status is displayed to users
2. Message History
    - Messages are loaded from history on connection
    - Prevents data loss during disconnections
3. Typing Indicator Optimization
    - Uses a 2-second timeout to prevent excessive events
    - Clears timeout on each keypress to debounce
4. Error Handling
    - Callbacks are used for operation confirmations
    - System messages display errors to users


Railway Cloud Platform
Node.js Application Container
Node.js Applicantion Container
Application Components (Production)
Express.js Server
User Service
Socket.IO Server
Message Service
Uses
Remote User (Person)
Web Browser
HTTPS/WSS connects to production URL
Deploys
Devloys to
Develops
Developer (Person)
Deploys to
CRUD operations
CRUD operations
Docker Desktop (Local)
Node.js Application Container
MongoDB Container
Application Components (Development)
Express.js Server
User Service
Socket.IO Server
Message Service
Node.js Application Container
Develops
Deploys to
Developer (Person)

Chat Application User Guide

Getting Started

Accessing the Chat Application

1. Development/Local Access:
    - Access the application via http://localhost:3000 (or your configured port)
    - Make sure Docker Desktop is running
2. Production Access:
    - Access the application via https://chat-app-production-7ff3.up.railway.app

Registration

1. When you first open the application, you'll be prompted to enter a username
2. Enter your desired username and click OK
3. Your username will be saved for future sessions

Interface Overview

The chat interface is divided into two main sections:

Left Sidebar

- Shows the list of currently online users
- Displays your connection status at the top
- Updates in real-time as users join or leave

Main Chat Area

- Displays all messages in the conversation
- Your messages appear on the right side (blue)
- Others' messages appear on the left side (gray)
- System messages appear centered with italic text

Sending Messages

1. Type your message in the input field at the bottom
2. Press Enter or click the send button (paper airplane icon)
3. Your message will appear in the chat and be visible to all users

Message Features

Message Information

Each message displays:

- Sender's name (for others' messages)
- Time the message was sent
- Message content

Editing Your Messages

1. Hover over one of your messages
2. Click the "Edit" button that appears
3. A prompt will open with your original message
4. Edit the text and click OK
5. Your message will be updated for all users

Deleting Your Messages

1. Hover over one of your messages
2. Click the "Delete" button that appears
3. Confirm the deletion
4. The message will be removed for all users

Real-Time Features

Typing Indicators

When someone is typing:

1. A typing indicator will appear at the bottom of the chat
2. It shows "[Username] is typing..."
3. The indicator disappears when typing stops

Online Status

1. The sidebar shows all currently connected users
2. Your name appears with "(You)" next to it
3. The list updates in real-time as users join or leave

Connection Status

1. The connection status appears at the top of the sidebar
2. "Connected" indicates a successful connection to the server
3. "Disconnected" indicates connection loss
4. "Connecting..." appears when attempting to reconnect

Mobile Usage

The chat interface is responsive and works on mobile devices:

- On smaller screens, the chat area takes priority
- To see online users on mobile, tap/swipe from the left edge

Troubleshooting

Connection Issues

If you experience connection problems:

1. Check your internet connection
2. Refresh the page
3. Clear browser cache if necessary
4. The application will attempt to reconnect automatically

Message Not Sending

If your messages aren't sending:

1. Check the connection status indicator
2. Ensure your message isn't empty
3. Refresh the page if problems persist

Missing Username

If you're prompted for a username again:

1. Enter your username
2. Check if your browser has cookies/local storage disabled
3. If using private/incognito mode, your username won't be remembered

Privacy & Security Notes

- Messages are not end-to-end encrypted
- Usernames are stored in your browser's local storage
- All messages are visible to everyone in the chat
- Don't share sensitive personal information in the chat