**Using RabbitMQ and WebSockets for Real-Time Chat in Microservices**

Student: Ionescu Anda

Group: 258

**Introduction**

Microservices have revolutionized application development by enabling modular, scalable, and maintainable architectures. However, implementing real-time communication, such as chat functionality, in a microservices-based system can be challenging. This tutorial demonstrates how to implement a **real-time chat system** using **RabbitMQ** as a message broker and **WebSockets** for real-time communication.

RabbitMQ is a robust message broker that enables asynchronous communication between microservices, while WebSockets provide a persistent, bidirectional communication channel between the client and server. Together, they form a powerful solution for real-time features like chat.

**Prerequisites**

* **Node.js** (version 16 or higher)
* **NestJS CLI** (for backend development)
* **Angular CLI** (for frontend development)
* **Docker** (for running RabbitMQ and MongoDB)
* **Postman** (for testing APIs)

**Step 1: Set Up RabbitMQ Using Docker**

RabbitMQ is used as the message broker to handle chat messages.

1. **Add RabbitMQ to docker-compose.yml**:  
   Update your docker-compose.yml file to include RabbitMQ:

rabbitmq:

image: rabbitmq:management

container\_name: rabbitmq

ports:

- "5672:5672" # AMQP port

- "15672:15672" # Management UI port

environment:

RABBITMQ\_DEFAULT\_USER: guest

RABBITMQ\_DEFAULT\_PASS: guest

networks:

- app-network

1. **Start RabbitMQ**:  
   Run the following command to start RabbitMQ:

bash

Copy

docker-compose up -d rabbitmq

1. **Access RabbitMQ Management UI**:  
   Open your browser and navigate to http://localhost:15672. Use the credentials guest/guest to log in.

**Step 2: Implement Chat Microservice in NestJS**

The chat microservice will handle sending and receiving messages using RabbitMQ and WebSockets.

1. **Install Required Dependencies**:  
   Install the necessary packages for RabbitMQ and WebSockets:

npm install @nestjs/websockets @nestjs/microservices amqplib

1. **Create the Chat Gateway**:  
   The ChatGateway will handle WebSocket connections and events.

import { WebSocketGateway, WebSocketServer, SubscribeMessage, MessageBody } from '@nestjs/websockets';

import { Server } from 'socket.io';

@WebSocketGateway()

export class ChatGateway {

@WebSocketServer()

server: Server;

@SubscribeMessage('message')

handleMessage(@MessageBody() data: string): void {

this.server.emit('message', data); // Broadcast the message to all clients

}

}

1. **Create the Chat Service**:  
   The ChatService will interact with RabbitMQ to send and receive messages.

import { Injectable } from '@nestjs/common';

import \* as amqp from 'amqplib';

@Injectable()

export class ChatService {

private channel: amqp.Channel;

async connect() {

const connection = await amqp.connect('amqp://localhost:5672');

this.channel = await connection.createChannel();

await this.channel.assertQueue('chat\_messages');

}

async sendMessage(message: string) {

this.channel.sendToQueue('chat\_messages', Buffer.from(message));

}

async consumeMessages(callback: (message: string) => void) {

this.channel.consume('chat\_messages', (msg) => {

if (msg) {

callback(msg.content.toString());

this.channel.ack(msg);

}

});

}

}

1. **Integrate RabbitMQ with WebSockets**:

Update the ChatGateway to use the ChatService for sending and receiving messages:

import { WebSocketGateway, WebSocketServer, SubscribeMessage, MessageBody } from '@nestjs/websockets';

import { Server } from 'socket.io';

import { ChatService } from './chat.service';

@WebSocketGateway()

export class ChatGateway {

@WebSocketServer()

server: Server;

constructor(private readonly chatService: ChatService) {

this.chatService.connect();

this.chatService.consumeMessages((message) => {

this.server.emit('message', message); // Broadcast messages from RabbitMQ

});

}

@SubscribeMessage('message')

handleMessage(@MessageBody() data: string): void {

this.chatService.sendMessage(data); // Send messages to RabbitMQ

}

}

**Step 2: Implement Chat Frontend in Angular**

The frontend will use WebSockets to send and receive messages in real-time.

1. **Install WebSocket Library**:  
   Install the socket.io-client library: npm install socket.io-client
2. **Create a Chat Service in Angular**:  
   The ChatService will handle WebSocket connections.

import { Injectable } from '@angular/core';

import { io, Socket } from 'socket.io-client';

@Injectable({

providedIn: 'root',

})

export class ChatService {

private socket: Socket;

constructor() {

this.socket = io('http://localhost:3000'); // Connect to the backend

}

sendMessage(message: string) {

this.socket.emit('message', message); // Send a message

}

onMessage(callback: (message: string) => void) {

this.socket.on('message', callback); // Listen for messages

}

}

1. **Create a Chat Component**:  
   The ChatComponent will display messages and allow users to send new messages.

import { Component, OnInit } from '@angular/core';

import { ChatService } from './chat.service';

@Component({

selector: 'app-chat',

template: `

<div>

<div \*ngFor="let message of messages">{{ message }}</div>

<input [(ngModel)]="newMessage" />

<button (click)="sendMessage()">Send</button>

</div>

`,

})

export class ChatComponent implements OnInit {

messages: string[] = [];

newMessage: string = '';

constructor(private chatService: ChatService) {}

ngOnInit() {

this.chatService.onMessage((message) => {

this.messages.push(message); // Display new messages

});

}

sendMessage() {

this.chatService.sendMessage(this.newMessage); // Send a new message

this.newMessage = '';

}

}

**Step 4: Test the Chat System**

1. Start the Backend:  
   Run the NestJS application: npm run start
2. **Start the Frontend**:  
   Run the Angular application: ng serve
3. **Open the Chat Interface**:  
   Navigate to http://localhost:4200 and test the chat functionality by sending and receiving messages in real-time.

Conclusion

This tutorial demonstrated how to implement a real-time chat system using RabbitMQ and WebSockets in a microservices architecture. RabbitMQ handles message queuing, while WebSockets enable real-time communication between the client and server. This solution is scalable, modular, and can be extended to support additional features like notifications or event streaming.

The complete code for this tutorial is available in the following public repository: