**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»**

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт по лабораторній роботі № 1

Протокол WebSocket. Використання Socket.io

для розробки чат-додатків

назва лабораторної роботи

з дисципліни: «Основи WEB технологій»

Студент: Маланічев Денис Андрійович

Група: ІС-13

Київ, 2024

**Тема:** Протокол WebSocket. Використання Socket.io для розробки чат-додатків

**Хід роботи**

Наш чат матиме клієнт-серверну архітектуру. Розробимо серверну частину у вигляді консольної програми на мові C# і клієнтський інтерфейс у вигляді WEB-програми з допомогою Angular. Сервер буде, власне, перевіряти усі запити на підключення до WebSocket:

public static async Task Main()

    {

        var httpListener = new HttpListener();

        httpListener.Prefixes.Add("http://localhost:5000/");

        httpListener.Start();

        Console.WriteLine("WebSocket server started at ws://localhost:5000");

        while (true)

        {

            HttpListenerContext context = await httpListener.GetContextAsync();

            if (context.Request.IsWebSocketRequest)

            {

                HttpListenerWebSocketContext wsContext = await context.AcceptWebSocketAsync(null);

                WebSocket webSocket = wsContext.WebSocket;

                Console.WriteLine("Client connected!");

                \_clients.Add(webSocket);

                \_ = Task.Run(() => HandleClientAsync(webSocket));

            }

            else

            {

                context.Response.StatusCode = 400;

                context.Response.Close();

            }

        }

    }

Як видно з коду вище, ми в нескінченному циклі перевіряємо, чи нема запитів на підключення. Проте даний цикл не виконується постійно – асинхронний метод GetContextAsync() поверне значення лише коли новий клієнт надіслав запит на підключення. Далі ми перевіряємо, чи є запит коректним і якщо запит некоректний, то закриваємо контекст даного запиту та повертаємо 400 статус код. Якщо ж запит на підключення був коректним, то ми дозволяємо підключення, додаємо створений сокет в список клієнтів і починаємо асинхронно опрацьовувати запити до даного клієнта.

Тепер створимо модель наших повідомлень. Вона міститиме наступні поля: UserId, Message та TimeSend.

public class MessageModel

{

    public int UserId { get; set; }

    public string Message { get; set; } = null!;

    public string TimeSend { get; set; } = null!;

}

Тепер розглянемо безпосередньо обробку повідомлень. Першочергово нам необхідно створити буфер. Який зберігатиме передані в повідомленні дані:

var buffer = new byte[10240];

Далі, поки даний сокет має стан «відкритий» ми очікуємо надходження до нього повідомлень. Коли таке повідомлення надійшло, ми перевіряжмо його тип. Якщо тип повідомлення Close, тобто підключення до даного сокета було закрито, ми виконуємо закриття створеного на сервері сокету та видаляємо його з списку клієнтів. Обов’язково робимо це в конструкції lock, адже в іншому потоці можливе одночасне звертання до цього ж сокету.

while (webSocket.State == WebSocketState.Open)

            {

                WebSocketReceiveResult result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer), CancellationToken.None);

                // If the client wants to close the WebSocket

                if (result.MessageType == WebSocketMessageType.Close)

                {

                    Console.WriteLine("Client disconnected.");

                        await webSocket.CloseAsync(WebSocketCloseStatus.NormalClosure, "Closing", CancellationToken.None);

                    lock(new object())

                    {

                        \_clients.Remove(webSocket);  // Remove client from list

                    }

                    break;

                }

                string clientMessageJson = Encoding.UTF8.GetString(buffer, 0, result.Count);

                MessageModel messageModel = null!;

                try

                {

                    messageModel = JsonConvert.DeserializeObject<MessageModel>(clientMessageJson)!;

                    if (messageModel is null)

                    {

                        Console.WriteLine("Error: Received empty or invalid message.");

                        continue;

                    }

                    Console.WriteLine($"User (ID: {messageModel.UserId}) - {messageModel.Message} - (at {messageModel.TimeSend})");

                    // Broadcast the message to all connected clients

                    await BroadcastMessageAsync(webSocket, messageModel);

                }

                catch (Exception jsonEx)

                {

                    Console.WriteLine("Error parsing JSON message: " + jsonEx.Message);

                }

            }

Якщо повідомлення не закривало зв’язку, а навпаки, надсилало певний текст, ми вичитуємо його дані в буфер і десереалізуємо їх в об’єкт створеної моделі, та передаємо всім користувачам чату:

private static async Task BroadcastMessageAsync(WebSocket sender, MessageModel message)

    {

        foreach (var client in \_clients)

        {

            if (client != sender && client.State == WebSocketState.Open)

            {

                byte[] responseBytes = Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(message));

                await client.SendAsync(new ArraySegment<byte>(responseBytes), WebSocketMessageType.Text, true, CancellationToken.None);

            }

        }

    }

Тепер перейдемо до клієнтського застосунку. Він складається з всього 1 модуля, 1 сервісу та 2 компонентів.

Головний компонент app.component.ts у момент свого створення випадковим чином генерує ID користувача, інжектує WebSocketService, підписується на оновлення даного сокету через observable.

export class AppComponent implements OnInit, OnDestroy {

  public messages: MessageWithOwnerModel[] = [];

  public message: string = '';

  public userId: number = Math.floor(Math.random() \* (99999 - 10000) + 10000);

  private socketSubscription!: Subscription;

  constructor(private webSocketService: WebSocketService) {}

  ngOnInit(): void {

    this.socketSubscription = this.webSocketService

      .connect('ws://localhost:5000')

      .subscribe(

        (message: string) => {

          const messageData: MessageModel = JSON.parse(message);

          this.messages.push({ ...messageData, OwnMessage: false });

        },

        (error) => {

          console.error('WebSocket error', error);

        }

      );

  }

Окрім підписки на повідомлення, даний компонент також має методи для відправки повідомлень, а в момент свого закриття компонент закриває всі підключення:

 sendMessage(): void {

    if (this.message.trim()) {

      const messageData: MessageModel = {

        UserId: this.userId,

        Message: this.message,

        TimeSend: new Date().toISOString(),

      };

      const messageJson = JSON.stringify(messageData);

      this.messages.push({ ...messageData, OwnMessage: true });

      this.webSocketService.connect('ws://localhost:5000').next(messageJson);

      this.message = '';

    }

  }

  ngOnDestroy(): void {

    this.socketSubscription.unsubscribe();

    this.webSocketService.disconnect();

  }

Як видно з наступного коду, ми виводимо всі отримані повідомлення, а до кнопки Send додано подію, що викликає метод sendMessage, коли кнопка натиснута.

<div class="chat-container">

  <h2>WebSocket Chat</h2>

  <div class="messages-container" #messagesContainer>

    <div \*ngFor="let msg of messages">

      <app-message [messageModel]="msg" />

    </div>

  </div>

  <div class="input-container">

    <input [(ngModel)]="message" type="text" placeholder="Type your message..." />

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

  </div>

</div>

Власне WebSocket сервіс має наступний вигляд:

export class WebSocketService {

  private socket!: WebSocket;

  private subject!: Subject<string>;

  constructor() { }

  public connect(url: string): Subject<string> {

    if (!this.subject) {

      this.subject = this.create(url);

      console.log('Successfully connected: ' + url);

    }

    return this.subject;

  }

  private create(url: string): Subject<string> {

    this.socket = new WebSocket(url);

    const observable = new Observable<string>(observer => {

      this.socket.onmessage = (event) => observer.next(event.data);

      this.socket.onerror = (event) => observer.error(event);

      this.socket.onclose = () => observer.complete();

      return () => {

        this.socket.close();

      };

    });

    const observer = {

      next: (data: string) => {

        if (this.socket.readyState === WebSocket.OPEN) {

          this.socket.send(data);

        }

      }

    };

    return Subject.create(observer, observable);

  }

  public disconnect() {

    if (this.socket) {

      this.socket.close();

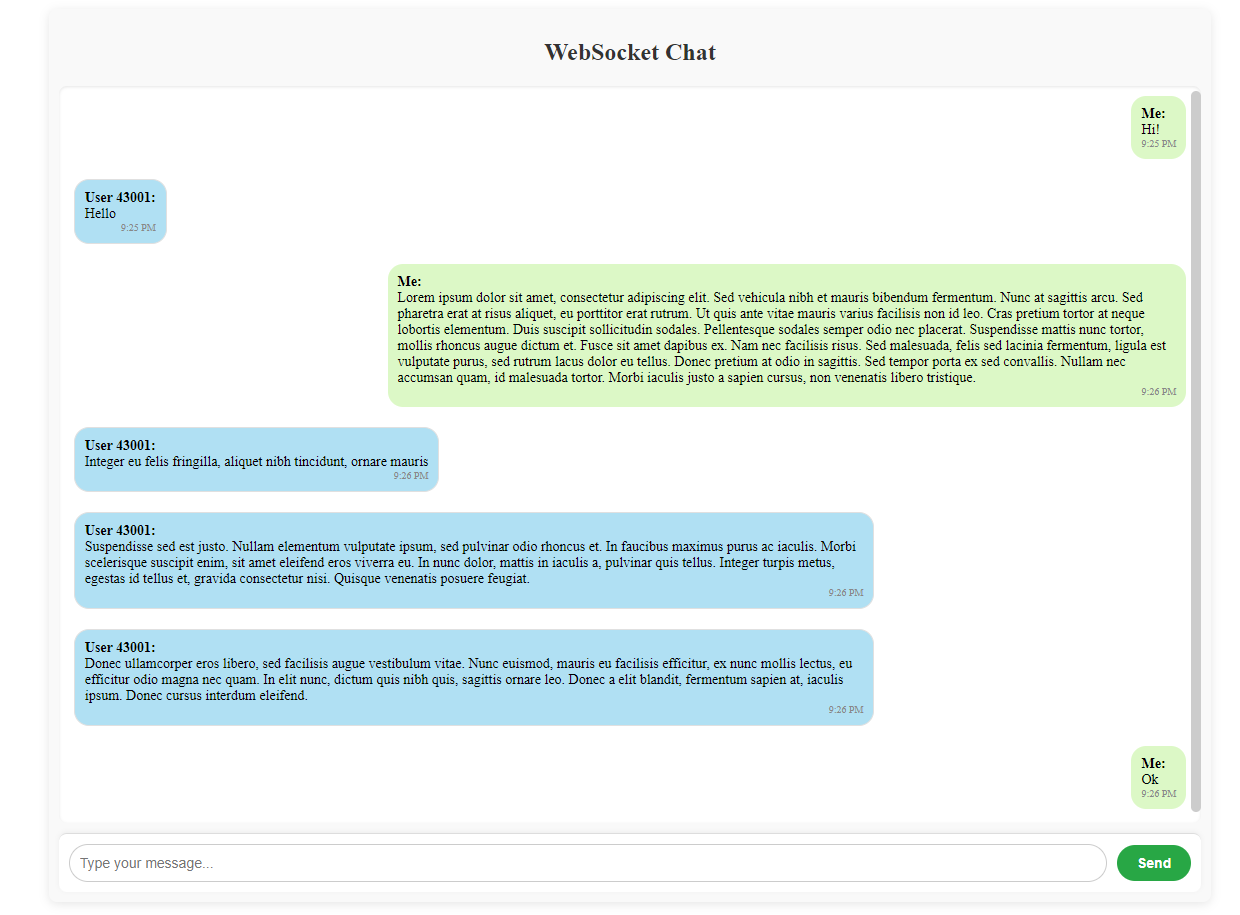
    }

  }

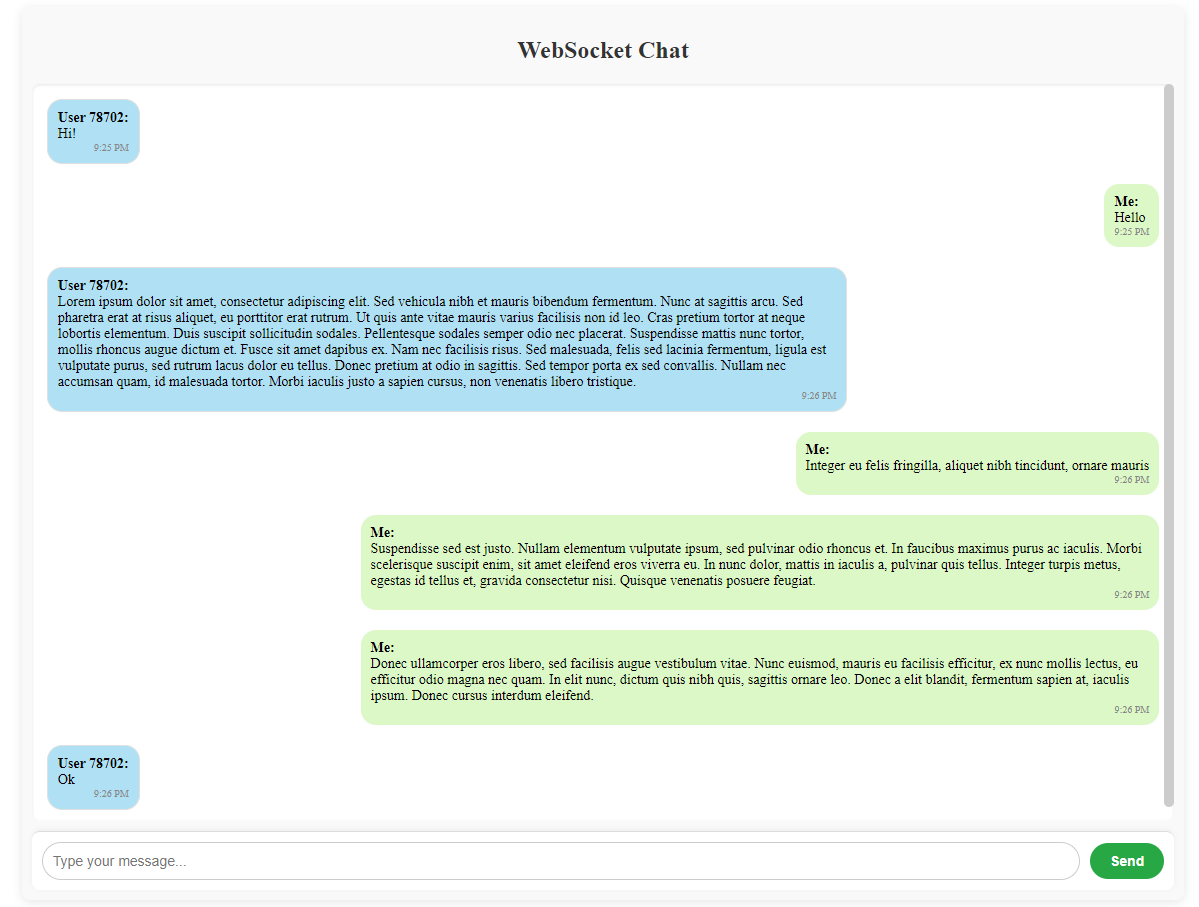
}

Його задача – створити підключення до сервера та відправити повідомлення.

Додавши темплейт та стилі до компонентів маємо наступний вигляд нашого чату:



Відповідно другий користувач матиме наступний вигляд чату:



Посилання GitHub репозиторій: https://github.com/DenysMalanichev/WebSocketChat/tree/master