**Лабораторна робота №4.**

**Тема: Додаток для спілкування в реальному часі**

**Виконавець: Гончаренко Микита Тв-12**

Створіть програму для чату в режимі реального часу, яка включає такі функції, як обмін приватними повідомленнями онлайн або перегляд історії повідомлень.

Спочатку треба реалізувати сервер

**1. Імпорт бібліотек:**

use tokio::{ net::TcpListener, sync::{mpsc, Mutex} };

use tokio\_tungstenite::{ accept\_async, tungstenite::Message };

use std::sync::Arc;

use futures::{ stream::StreamExt, SinkExt };

* **Tokio**: асинхронний фреймворк для роботи з мережевими з'єднаннями.
* **Tokio-Tungstenite**: бібліотека для роботи з WebSocket протоколом.
* **Futures**: дозволяє працювати з потоками даних.
* **Arc** і **Mutex**: забезпечують потокобезпеку для спільного доступу до даних.

**2. Визначення типу Sender:**

type Sender = mpsc::UnboundedSender<Message>;

* Тип Sender представляє канал для надсилання повідомлень клієнтам.

**3. Структура ChannelManager:**

struct ChannelManager {

channels: Arc<Mutex<std::collections::HashMap<String, Vec<Sender>>>>,

}

* Зберігає список кімнат (HashMap), де:
  + Ключ: назва кімнати (рядок).
  + Значення: список відправників (Sender), прив'язаних до кімнати.

4. Методи ChannelManager:

* new: створює новий екземпляр менеджера кімнат.
* get\_or\_create\_channel: перевіряє, чи існує кімната; якщо ні — створює її.
* add\_sender\_to\_channel: додає клієнта (відправника) до кімнати.
* remove\_sender\_from\_channel: видаляє клієнта з кімнати.
* broadcast: надсилає повідомлення всім клієнтам у певній кімнаті.

**5. Обробка підключення клієнта:**

async fn handle\_connection(

stream: tokio::net::TcpStream,

addr: std::net::SocketAddr,

channel\_manager: Arc<ChannelManager>,

)

* Обробляє нове WebSocket-з'єднання.
* Основні етапи:

1. WebSocket-з'єднання: приймає з'єднання через accept\_async.
2. Розділення потоків: ws\_stream.split() розділяє потоки читання (read) і запису (write).
3. Обробка команд від клієнта:

* CREATE\_ROOM:<назва> — створити кімнату.
* JOIN\_ROOM:<назва> — приєднатися до кімнати.
* LEAVE\_ROOM:<назва> — покинути кімнату.
* ROOM\_MSG:<кімната>:<користувач>:<повідомлення> — надіслати повідомлення в кімнату.

**6. Метод start\_server:**

pub async fn start\_server() -> Result<(), Box<dyn std::error::Error>> {

let listener = TcpListener::bind("127.0.0.1:8080").await?;

println!("webSocket server Started at ws://127.0.0.1:8080");

let channel\_manager = Arc::new(ChannelManager::new());

while let Ok((stream, addr)) = listener.accept().await {

let channel\_manager: Arc<ChannelManager> = channel\_manager.clone();

tokio::spawn(handle\_connection(stream, addr, channel\_manager));

}

Ok(())

}

* Створює TCP-сервер на порту 8080.
* Для кожного нового підключення створюється асинхронний таск (tokio::spawn), який обробляє з'єднання клієнта через функцію handle\_connection.

**Як працює сервер:**

1. Клієнт підключається до сервера через WebSocket (ws://127.0.0.1:8080).
2. Клієнт може виконувати наступні дії:

* Створювати кімнати.
* Приєднуватися до кімнат.
* Надсилати повідомлення в кімнати.
* Покидати кімнати.

1. Сервер обробляє повідомлення та пересилає їх іншим клієнтам у кімнаті.

**Тепер оглянемо структуру клієнта:**

**1. Імпорти:**

use tokio::sync::{mpsc, Mutex};

use tokio\_tungstenite::{

connect\_async,

tungstenite::Message,

};

use futures::{

stream::StreamExt,

SinkExt,

};

use std::{

error::Error, io::{self, Write}, sync::Arc

};

* **Tokio**: забезпечує асинхронну обробку.
* **Tokio-Tungstenite**: для підключення та роботи з WebSocket.
* **Futures**: для потоків обміну даними між клієнтом і сервером.
* **std::io**: для роботи з введенням/виведенням даних від користувача.

**2. Підключення до сервера:**

let (ws\_stream, \_) = connect\_async("ws://127.0.0.1:8080").await?;

println!("Connection to WebSocket server");

* Клієнт встановлює WebSocket-з'єднання із сервером за адресою ws://127.0.0.1:8080.

**3. Розділення потоків:**

let (mut write, mut read) = ws\_stream.split();

* Потоки для читання (read) і запису (write) від сервера розділяються для асинхронної обробки.

**4. Створення каналу для надсилання повідомлень:**

let (tx, mut rx) = mpsc::unbounded\_channel::<Message>();

* tx: використовується для надсилання повідомлень до сервера.
* rx: отримує ці повідомлення.

**5. Обробка надсилання і отримання повідомлень:**

* **Відправлення повідомлень серверу**:

tokio::spawn(async move {

while let Some(msg) = rx.recv().await {

write.send(msg).await.expect("Failed to send message");

}

});

* + У новому таску клієнт відправляє повідомлення серверу через канал.
* **Отримання повідомлень від сервера**:

tokio::spawn(async move {

while let Some(msg) = read.next().await {

match msg {

Ok(Message::Text(text)) => {

let mut msgs = message.lock().await;

if !text.starts\_with("ROOM\_MSG:") {

msgs.push(text.clone());

println!("{}", text);

}

}

Err(e) => eprintln!("Error receiving message: {}", e),

\_ => println!("Received an unknown message type"),

}

}

});

* + У іншому таску клієнт приймає текстові повідомлення і зберігає їх у пам'ять (message).

**6. Інтерактивний інтерфейс користувача:**

* **Введення імені користувача**:

print!("Enter your username: ");

io::stdout().flush()?;

let mut username = String::new();

io::stdin().read\_line(&mut username)?;

let username = username.trim().to\_string();

* + Запитує ім’я користувача.
* **Створення або приєднання до кімнати**:

if input.starts\_with("CREATE ") {

current\_room = input[7..].to\_string();

tx.send(Message::Text(format!("CREATE\_ROOM:{}", current\_room)))

.expect("Failed to send message");

} else if input.starts\_with("JOIN ") {

current\_room = input[5..].to\_string();

tx.send(Message::Text(format!("JOIN\_ROOM:{}", current\_room)))

.expect("Failed to send message");

}

* + Користувач може:
    - Створити кімнату (CREATE <room\_name>).
    - Приєднатися до кімнати (JOIN <room\_name>).
* **Вивід попередніх повідомлень**:

rust

Копіювати код

let msgs = message.lock().await;

for msg in msgs.iter() {

println!("{}", msg);

}

* + Показує історію повідомлень після приєднання.

**7. Чат у кімнаті:**

loop {

print!("{} > ", username);

io::stdout().flush()?;

let mut message = String::new();

io::stdin().read\_line(&mut message)?;

let message = message.trim();

if message == "/leave" {

tx.send(Message::Text(format!("LEAVE\_ROOM: {}", current\_room)))

.expect("Failed to send message");

break;

}

tx.send(Message::Text(format!("ROOM\_MSG:{}:{}:{}", current\_room, username, message)))

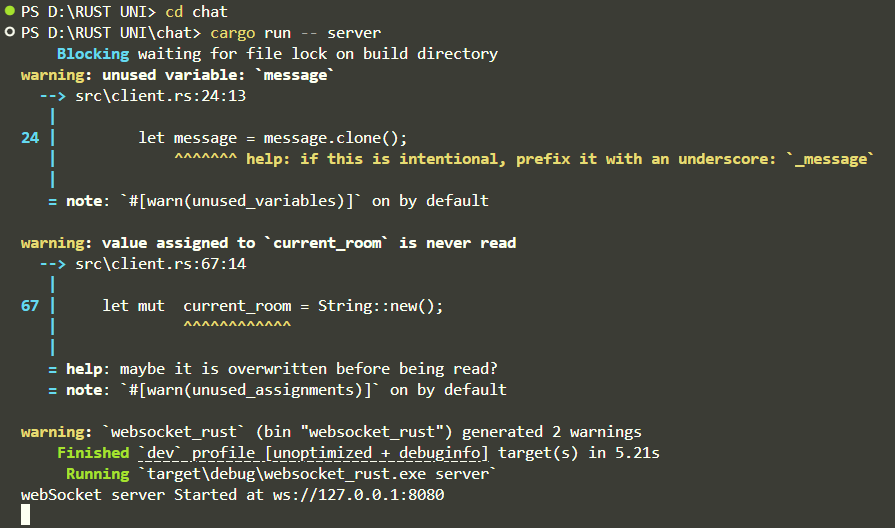
.expect("Failed to send message");

}

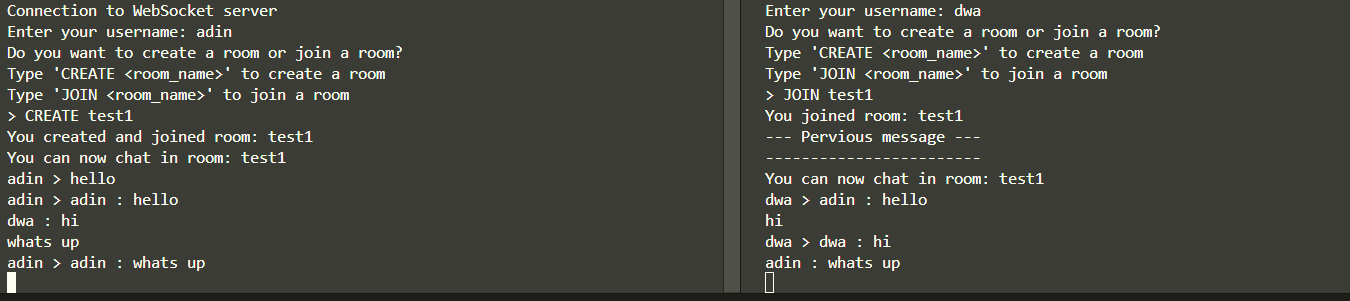
* Після підключення до кімнати користувач може:
  + Відправляти повідомлення в кімнату.
  + Покидати кімнату командою /leave.

**Приклад виконання:**

Запускаємо сервер



Запускаємо двох клієнтів для тесту



Бачимо що все працює і виводяться повідомлення

Висновок:

У ході виконання роботи було розроблено асинхронну клієнт-серверну систему для текстового чату з підтримкою WebSocket-з’єднань. Система дозволяє створювати і приєднуватися до окремих кімнат для обміну повідомленнями між користувачами в реальному часі.