Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота *№9*

із дисципліни ***«Технології розробки програмного забезпечення»***

Тема: ***«Взаємодія компонентів системи»***

Виконав:

Студент групи ІА-34

Ковальчук Станіслав

Перевірив:

асистент кафедри ІСТ

Мягкий Михайло Юрійович

Київ — 2025

Тема: Взаємодія компонентів системи.

Мета: Вивчити види взаємодії додатків (Client-Server, Peer-to-Peer, Serviceoriented Architecture), та реалізувати в проєктованій системі одну із архітектур..

Застосунок: Office communicator (strategy, adapter, abstract factory, bridge, composite, client-server)

Мережевий комунікатор для офісу повинен нагадувати функціонали програми Skype з можливостями голосового / відео / конференц-зв'язку, відправки текстових повідомлень і файлів (можливо, оффлайн), веденням організованого списку груп / контактів.

Короткі теоретичні відомості:

1. Клієнт-серверна архітектура

Це модель розподіленої системи, де завдання розподіляються між постачальниками послуг (**серверами**) та замовниками (**клієнтами**).

- **Тонкий клієнт:** Більшість операцій виконується на сервері. Клієнтська частина відповідає лише за відображення інтерфейсу (наприклад, веб-браузер). Це зменшує вимоги до заліза користувача, але підвищує навантаження на сервер.
- **Товстий клієнт:** Основна логіка обробки даних реалізована на стороні користувача. Сервер використовується переважно для зберігання даних та автентифікації (наприклад, десктопні версії Viber або Evernote).
- **SPA (Single Page Application):** Проміжний варіант, де інтерфейс завантажується один раз, а далі клієнт обмінюється з сервером лише даними (JSON/XML).

Рівні взаємодії:

1. **Презентаційний рівень (Client):** UI та логіка взаємодії.
2. **Проміжний рівень (Middleware):** Спільні класи, API-прослойки, обробка запитів.

3. **Серверний рівень (Backend):** Бізнес-логіка та робота з базами даних.

2. P2P-архітектура (Peer-to-Peer)

Мережева модель, де кожен вузол є рівноправним учасником: він одночасно виступає і клієнтом, і сервером.

- **Принципи:** Децентралізація, висока відмовостійкість (вихід з ладу одного вузла не зупиняє систему) та спільне використання ресурсів (диск, CPU).
- **Застосування:** BitTorrent, блокчейн (Bitcoin), IP-телефонія.
- **Виклики:** Складність гарантування безпеки, синхронізація станів між усіма вузлами та ефективний пошук ресурсів у великих мережах.

3. Сервіс-орієнтована архітектура (SOA)

Модульний підхід, де система будується з набору слабо пов'язаних (loosely coupled) сервісів, що взаємодіють через стандартні протоколи.

- **Ключові риси:**
    - **Слабке зв'язування:** Сервіси незалежні; зміна одного не повинна руйнувати роботу іншого.
    - **ESB (Enterprise Service Bus):** Централізована шина даних, яка допомагає сервісам «спілкуватися» між собою.
    - **Legacy-інтеграція:** Дозволяє створювати «обгортки» (adapters) навколо застарілих систем, надаючи їм сучасний інтерфейс.

4. Мікросервісна архітектура

Еволюційний розвиток SOA, де додаток розбивається на максимально дрібні, автономні сервіси, кожен з яких відповідає за конкретну бізнес-функцію.

- **Автономність:** Кожен мікросервіс може мати власну базу даних та бути написаним на іншій мові програмування.

- **Взаємодія:** Зазвичай через легковажні протоколи (HTTP/REST, gRPC) або брокери повідомлень (RabbitMQ, Kafka).
- **Переваги:** Незалежне розгортання та масштабування. Якщо один сервіс вийде з ладу, решта системи продовжує працювати.

Хід роботи:

1. Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

   Даний програмний продукт реалізовано на основі **клієнт-серверної архітектури**. Це підтверджується чітким розподілом ролей між компонентами системи та організацією взаємодії через мережеві протоколи.

   1. Розподіл функціональних ролей

   Система розділена на дві незалежні частини з власними зонами відповідальності:

- **Клієнт (Frontend):** Представлений веб браузером користувача.
    - **Технологічний стек:** HTML (Index.cshtml), клієнтський JavaScript (chat.js) та API WebRTC.
    - **Функції:** Візуалізація інтерфейсу, обробка дій користувача (введення тексту, кліки), ініціація запитів до сервера. Клієнт є «тонким», оскільки не зберігає дані локально та не має прямого доступу до БД.
- **Сервер (Backend):** Реалізований на базі **ASP.NET Core** (WebApplication2).
    - **Технологічний стек:** C#, SignalR Hubs, Entity Framework Core.
    - **Функції:** Обробка HTTP-запитів через контролери, підтримання активних з’єднань, виконання бізнес-логіки, розсилка повідомлень у реальному часі та координація медіапотоків.

2. Мережева взаємодія та протоколи

Взаємодія між компонентами не є прямою, а здійснюється виключно через мережу:

- **HTTP/HTTPS:** Для завантаження статичного контенту та сторінок.
- **SignalR (WebSockets):** Для забезпечення миттєвого обміну повідомленнями.
- **WebRTC Signaling:** Сервер виступає посередником для встановлення прямого з'єднання між клієнтами.

Ключова ознака: Клієнт ніколи не працює з ресурсами системи (наприклад, файлами чи БД) напряму — кожен запит проходить через шар безпеки та логіки сервера.

3. Централізація даних та безпека

Вся важлива інформація зберігається централізовано на стороні сервера в базі даних **SQLite**:

- Доступ до даних реалізовано через серверні репозиторії (EfMessageRepository).
- Сервер контролює цілісність даних та автентифікацію користувачів.
- Такий підхід дозволяє багатьом клієнтам мати актуальну та синхронізовану інформацію одночасно.

4. Масштабованість та незалежність

Архітектура забезпечує автономність компонентів:

1. **Багатокористувацький доступ:** Один сервер здатний одночасно обслуговувати велику кількість підключених клієнтів.
2. **Гнучкість розробки:** Можливість оновлювати серверний код (логіку збереження чи обробки) без необхідності переналаштування клієнтської частини у користувача.
3. **Ізоляція:** Збій на боці одного клієнта не впливає на роботу сервера або інших учасників мережі.

5. Реалізація взаємодії класів (на прикладі сценарію відправки повідомлення):

Для досягнення функціональної можливості "Миттєвий обмін повідомленнями" реалізовано ланцюжок взаємодії наступних класів системи:

Основні класи-учасники:

1. ChatHub (Mediator): Центральний клас сервера, що приймає сигнали від клієнта.
2. ChatMessage (Model): Клас-сутність, що передає дані (текст, автор, час).
3. EfMessageRepository (Repository): Відповідає за збереження даних у БД.
4. SignalRMessageSender (Bridge Implementation): Відповідає за технічну доставку повідомлення підключеним клієнтам.

Алгоритм взаємодії (Flow):

1. Ініціація: Клієнтський скрипт (chat.js) викликає серверний метод ChatHub.SendMessage(...) через протокол WebSocket.
2. Створення моделі: Клас ChatHub створює екземпляр класу ChatMessage, заповнюючи його отриманими даними.
3. Збереження (Persistance): ChatHub звертається до інтерфейсу IMessageRepository. Клас EfMessageRepository виконує запит до бази даних SQLite для збереження історії.
4. Відправка (Distribution):
    - ChatHub звертається до інтерфейсу IMessageSenderImplementation.
    - Клас SignalRMessageSender (конкретна реалізація) використовує IHubContext для розсилки повідомлення всім підписаним клієнтам у групі.
5. Візуалізація: JavaScript-клієнти отримують подію ReceiveMessage і оновлюють DOM-дерево сторінки.

Фрагмент коду взаємодії :

```
public async Task SendMessage(string channelId, string user, string message)
{
    Console.WriteLine($"TEXT MSG → {channelId} | {user}");

    var msg = new ChatMessage
    {
        ChannelId = channelId,
        SenderName = user,
        Content = message,
        SentAt = DateTime.UtcNow
    };

    await _repo.SaveAsync(msg);
    await _bridgeSender.SendAsync(msg);

    await Clients.Group(channelId).SendAsync(
        "ReceiveMessage",
        user,
        message,
        DateTime.Now.ToString("HH:mm")
    );
}
```

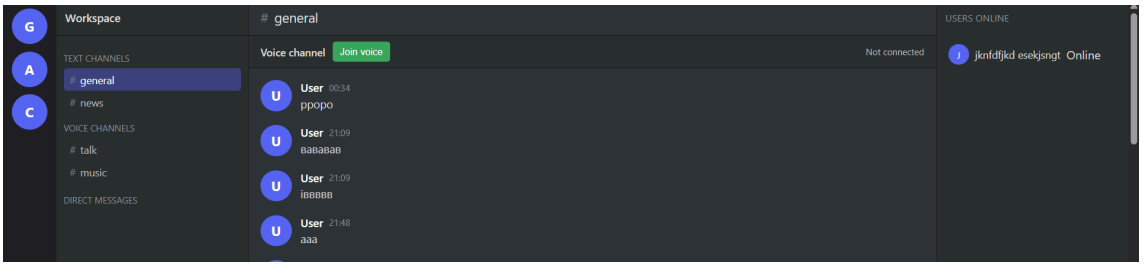

Рис. 9.1 – UI застосунку

2. Реалізувати функціонал для роботи в розподіленому оточенні відповідно до обраної теми.

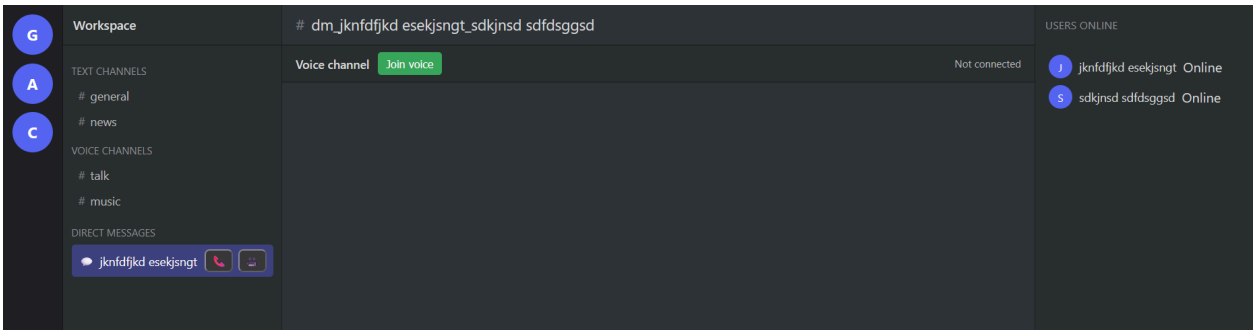

Рис. 9.2 – Вигляд приватних повідомлень

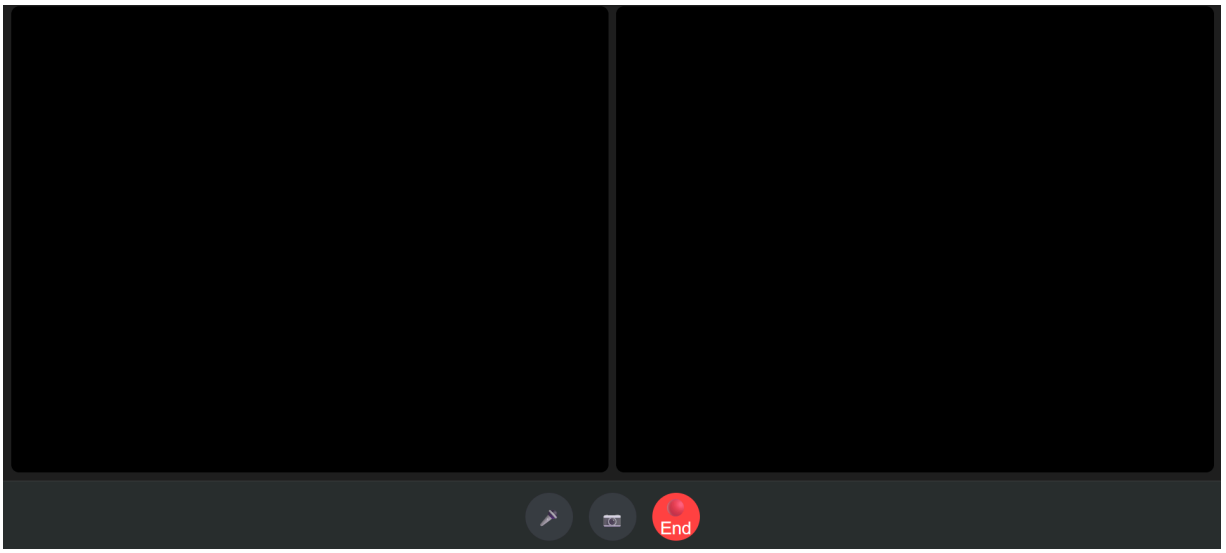

Рис. 9.3 – Вигляди голосової/відео кімніати.

Висновок: на даному лабораторному занятті я познайомився з тим, як взаємодіють додатки, і реалізував клієнт-серверну архітектуру, і отримав наступні переваги:

- **Централізоване управління:** Усі дані та бізнес-логіка зосереджені на сервері. Це дозволяє легко оновлювати систему, робити резервне копіювання даних та гарантувати, що всі користувачі бачать актуальну інформацію.
- **Безпека:** Клієнт не має прямого доступу до бази даних. Сервер виступає захисним бар'єром, який перевіряє права доступу та валідує кожен запит, що значно знижує ризик злому або втрати даних.
- **Масштабованість:** Архітектура дозволяє окремо покращувати потужність сервера (наприклад, додати пам'ять) або клієнтської частини без повної перебудови системи. Один сервер може ефективно обслуговувати тисячі різних клієнтів (веб, мобільні додатки).
- **Економічність ресурсів:** Завдяки використанню «тонких клієнтів» (браузерів), кінцевим користувачам не потрібні потужні комп'ютери — основні складні обчислення бере на себе продуктивний сервер.

Використання технології **SignalR** дозволило:

1. Уникнути прямого використання низькорівневих сокетів (TcpClient), замінивши їх на високорівневі абстракції (Hubs).
2. Забезпечити роботу через стандартні веб-порти (80/443), що спрощує розгортання в хмарі (відповідає вимогам SOA).
3. Реалізувати миттєву доставку повідомлень (Real-time) без необхідності постійного опитування сервера клієнтом.

Система успішно передає дані між клієнтом (браузер) та сервером (ASP.NET Core) у двосторонньому режимі.

Відповіді на контрольні питання:

1. Що таке клієнт-серверна архітектура?

Клієнт-серверна архітектура — це модель комп'ютерної мережі, в якій один комп'ютер (сервер) надає ресурси або послуги для інших комп'ютерів (клієнтів). Сервери зазвичай зберігають і обробляють дані, а клієнти запитують ці дані або послуги, щоб використовувати їх. Сервери і клієнти можуть знаходитися на різних машинах або в межах однієї мережі, і комунікація між ними здійснюється через мережу.

2. Розкажіть про сервіс-орієнтовану архітектуру (SOA).

Сервіс-орієнтована архітектура (SOA) — це стиль проектування програмного забезпечення, в якому система складається з незалежних, але взаємопов'язаних сервісів. Кожен сервіс є автономним, виконуючи конкретні бізнес-операції або обробку даних і може бути використаний іншими сервісами або клієнтами через стандартизовані інтерфейси. SOA дозволяє зменшити залежність між компонентами та забезпечує можливість масштабування і гнучкості при розробці програмних рішень.

3. Якими принципами керується SOA?

SOA базується на кількох ключових принципах:

- Автономія: кожен сервіс є незалежним, що дозволяє зменшити залежність між компонентами.
- Інтероперабельність: сервіси можуть взаємодіяти один з одним навіть якщо вони реалізовані на різних платформах або написані на різних мовах програмування.
- Стандартизовані інтерфейси: сервіси взаємодіють через стандартизовані протоколи і формати даних (наприклад, SOAP, REST).
- Легкість розширення: можливість додавати нові сервіси без великих змін в існуючій архітектурі.

- Повторне використання: сервіси можна повторно використовувати в різних програмах і контекстах.

4. Як між собою взаємодіють сервіси в SOA?

Сервіси в SOA взаємодіють через мережеві протоколи, як правило, використовуючи HTTP, SOAP або RESTful API. Кожен сервіс має чітко визначений інтерфейс, що дозволяє іншому сервісу або клієнту викликати його функціональність. Взаємодія відбувається через запити (наприклад, HTTP-запити або SOAP-повідомлення) та відповіді, які сервіси обробляють відповідно до визначених правил.

5. Як розробники взнають про існуючі сервіси і як робити до них запити?

Розробники можуть дізнатися про існуючі сервіси за допомогою:

- Реєстрації та каталогів сервісів: в організації може бути централізований реєстр всіх доступних сервісів, який містить інформацію про їх інтерфейси та способи взаємодії.
- WSDL (Web Services Description Language): для SOAP-сервісів доступна документація у форматі WSDL, яка описує доступні методи і параметри запитів.
- API документація: для RESTful сервісів існує документація, яка описує доступні кінцеві точки (endpoints), методи (GET, POST тощо) і структуру даних.

  Запити до сервісів виконуються через клієнтські бібліотеки або HTTP-клієнти, що використовують відповідні протоколи для виклику методів сервісів.

6. У чому полягають переваги та недоліки клієнт-серверної моделі?

Переваги:

- Централізація: дані і ресурси зберігаються на сервері, що дозволяє спрощувати управління і забезпечувати контроль за доступом.

- Безпека: оскільки дані зберігаються на сервері, їх можна захистити за допомогою централізованих методів аутентифікації і шифрування.
- Масштабованість: сервери можуть бути масштабовані для обробки великої кількості запитів.

Недоліки:

- Залежність від мережі: клієнт потребує доступу до мережі для взаємодії з сервером. Якщо сервер або мережа недоступні, клієнт не може працювати.
- Навантаження на сервер: якщо клієнти здійснюють багато запитів, сервер може стати перевантаженим.
- Обмеження на клієнтську обробку: клієнти часто мають обмежені можливості для обробки даних порівняно з серверами.

7. У чому полягають переваги та недоліки однорангової моделі взаємодії?

Переваги:

- Розподіленість: всі учасники (пірінги) рівні, і кожен з них може як надсилати, так і отримувати дані, що робить систему більш дистрибутивною.
- Без централізованої точки відмови: оскільки немає єдиного сервера, система не залежить від одного пункту.
- Гнучкість: учасники можуть додавати або видаляти себе з мережі без значних змін у загальній архітектурі.

Недоліки:

- Масштабованість: мережа може стати неефективною при великій кількості учасників, оскільки кожен піринг має обробляти запити від інших.
- Безпека: складніше забезпечити безпеку без централізованого контролю.

- Надмірне навантаження на учасників: кожен піринг відповідає за обробку запитів і зберігання даних, що може привести до перевантаження окремих учасників.

8. Що таке мікросервісна архітектура?

Мікросервісна архітектура — це стиль архітектури програмного забезпечення, де система розбивається на набір малих, незалежних сервісів, які взаємодіють між собою через добре визначені API. Кожен мікросервіс відповідає за певну бізнес-функціональність і може бути розгорнутий, оновлений та масштабований незалежно від інших сервісів.

9. Які протоколи використовуються для обміну даними в мікросервісній архітектурі?

Для обміну даними в мікросервісах зазвичай використовуються такі протоколи:

- HTTP/HTTPS для RESTful API.
- gRPC (Google Remote Procedure Call) для ефективних та швидких викликів між сервісами.
- AMQP або Kafka для асинхронної передачі повідомлень.
- WebSocket для двосторонньої комунікації в реальному часі.
- SOAP для більш традиційних сервісів з потребою в строгій схемі та безпеці.

10. Чи можна назвати підхід сервіс-орієнтованою архітектурою, коли ми в проєкті між шаром веб-контролерів та шаром доступу до даних реалізуємо шар бізнес-логіки у вигляді сервісів?

Так, цей підхід можна назвати сервіс-орієнтованою архітектурою (SOA), якщо шар бізнес-логіки реалізується як набір незалежних сервісів, які взаємодіють між собою через чітко визначені інтерфейси. SOA передбачає, що бізнес-логіка є автономною і може бути викликана з різних частин системи через стандартизовані методи, що відповідає принципам SOA.