Національний технічний університет України «Київський
політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

ЛАБОРАТОРНА РОБОТА №9

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Взаємодія компонентів системи»

Виконав:

студент групи ІА-33

Грицай Андрій

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.
Тема лабораторного циклу:

1. **Музичний програвач** (iterator, command, memento, facade, visitor, client-server)

Музичний програвач становить собою програму для програвання музичних файлів або відтворення потокової музики з можливістю створення, запам'ятовування і редагування списків програвання, перемішування/повторення (shuffle/repeat), розпізнавання різних аудіо-форматів, еквалайзер.

Теоретичні відомості

Клієнт-серверна архітектура (Client-Server Architecture)

Це модель взаємодії, що розділяє систему на постачальників ресурсів (сервери) та замовників послуг (клієнти). Використовується для централізації зберігання даних та бізнес-логіки, розвантажуючи клієнтську частину (тонкий клієнт) або розподіляючи навантаження (товстий клієнт). У Java реалізується через використання сокетів (Socket, ServerSocket) для низькорівневого зв'язку або через веб-фреймворки (Spring), де сервер обробляє HTTP-запити від клієнтських додатків.

Однорангова архітектура (Peer-to-Peer / P2P)

Це децентралізована мережева модель, де кожен вузол (пір) виступає одночасно і клієнтом, і сервером, маючи рівні права. Використовується для побудови стійких до відмов систем розподілу ресурсів без єдиної точки відмови. У Java реалізується шляхом створення додатків, які одночасно слухають вхідні з'єднання на певному порті та ініціюють вихідні запити до інших відомих вузлів мережі для обміну даними.

Сервіс-орієнтована архітектура (SOA)

Це модульний підхід, заснований на використанні слабко пов'язаних розподілених сервісів зі стандартизованими інтерфейсами. Використовується для інтеграції різнорідних систем та повторного використання бізнес-функціоналу через протоколи SOAP або REST. У Java реалізується через створення веб-сервісів (JAX-WS, JAX-RS), які можуть взаємодіяти через корпоративну сервісну шину (ESB) або прямі виклики, забезпечуючи незалежність компонентів.

Мікросервісна архітектура (Microservices)

Це стиль проєктування, при якому додаток розбивається на набір невеликих, незалежних служб, кожна з яких виконується у власному процесі.

Використовується для підвищення гнучкості, масштабованості та можливості незалежного розгортання окремих частин системи. У Java зазвичай реалізується за допомогою фреймворку Spring Boot, де кожен мікросервіс відповідає за окрему бізнес-функцію і комунікує з іншими через легковагі протоколи (HTTP/JSON, RabbitMQ).

Хід роботи

Для реалізації взаємодії компонентів системи було обрано клієнт-серверну архітектуру. Система розділена на дві незалежні частини: Клієнт (Client) та Сервер (Server), які взаємодіють між собою за допомогою мережевих сокетів (Java Sockets) через протокол TCP/IP.

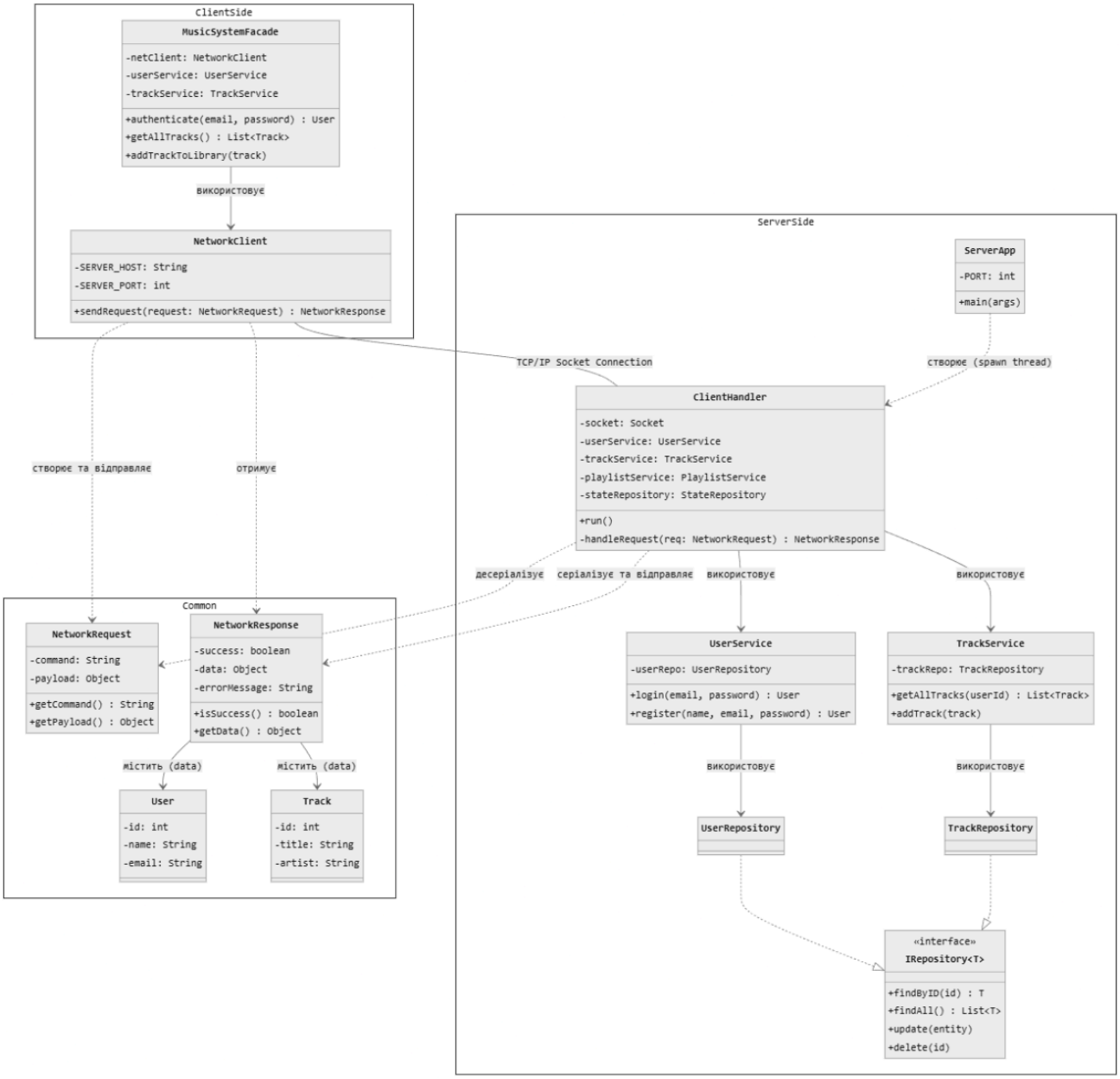

Рисунок 1 – Діаграма класів, яка представляє використання клієнт-серверної архітектури

На спроєктованій діаграмі класів виділено такі ключові елементи архітектури:

Клієнтська сторона:

- **MusicSystemFacade**: Виступає як єдина точка входу для клієнтської частини. Цей клас інкапсулює всю логіку взаємодії з мережею, надаючи контролерам простий інтерфейс (методи login, getAllTracks тощо). Він перетворює виклики методів у об'єкти NetworkRequest.
- **NetworkClient**: Відповідає за низькорівневе встановлення з'єднання з сервером, серіалізацію запитів та десеріалізацію відповідей (NetworkResponse).

Об'єкти передачі даних (DTO):

- **NetworkRequest**: Клас-обгортка, що містить назву команди (наприклад, "LOGIN", "ADD_TRACK") та корисне навантаження (payload).
- **NetworkResponse**: Клас-обгортка для відповіді сервера, що містить статус виконання, дані або повідомлення про помилку.

Серверна сторона:

- **ServerApp**: Точка входу сервера. Відкриває ServerSocket на визначеному порту та в нескінченному циклі очікує підключення клієнтів. Для кожного нового підключення створюється окремий потік обробки.
- **ClientHandler**: Потік, що обробляє запити від конкретного клієнта. Він десеріалізує NetworkRequest, визначає необхідну дію через конструкцію switch-case та звертається до відповідних сервісів.
- **Services** (UserService, TrackService, PlaylistService): Шар бізнес-логіки на сервері, який виконує реальну роботу та взаємодіє з базою даних через репозиторії.

Лістинг коду реалізованих класів системи

1.NetworkClient.

```
public class NetworkClient {

    private static final String SERVER_HOST = "localhost";

    private static final int SERVER_PORT = 8888;



    public NetworkResponse sendRequest(NetworkRequest request) {

        try (Socket socket = new Socket(SERVER_HOST, SERVER_PORT);

                                                ObjectOutputStream out = new
ObjectOutputStream(socket.getOutputStream());
```

```
                                                ObjectInputStream in = new
ObjectInputStream(socket.getInputStream())) {


            out.writeObject(request);

            return (NetworkResponse) in.readObject();


        } catch (Exception e) {

            e.printStackTrace();

                    return new NetworkResponse(false, "Помилка з'єднання: " +
e.getMessage());

        }

    }

}
```

2. MusicSystemFacade.

```
public class MusicSystemFacade {

    private final NetworkClient netClient;

    // ... ініціалізація сервісів ...


    public MusicSystemFacade() {

        this.netClient = new NetworkClient();

        // Приклад перевизначення методу сервісу для роботи через мережу

        this.trackService = new TrackService(null) {

            @Override

            public List<Track> getAllTracks(int userId) {

                // Відправка запиту на сервер

                                NetworkResponse resp = netClient.sendRequest(new
NetworkRequest("GET_USER_TRACKS", userId));

                // Обробка відповіді

                        return resp.isSuccess() ? (List<Track>) resp.getData() :
List.of();

            }
```

```
            @Override
            public void addTrack(Track track) {
                netClient.sendRequest(new NetworkRequest("ADD_TRACK", track));
            }
        };
        // ...
    }
}
```

3. ServerApp.

```
public class ServerApp {
    private static final int PORT = 8888;

    public static void main(String[] args) {
        // ... ініціалізація репозиторіїв та сервісів ...

        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            while (true) {
                Socket clientSocket = serverSocket.accept();
                // Створення та запуск потоку для обробки клієнта
                ClientHandler handler = new ClientHandler(
                        clientSocket,
                        userService,
                        trackService,
                        playlistService,
                        stateRepo
                );
                handler.start();
            }
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

4. ClientHandler.

```
public class ClientHandler extends Thread {
    // ... поля сервісів ...

    @Override
    public void run() {
        try (
                                                ObjectOutputStream out = new
ObjectOutputStream(socket.getOutputStream());
                                                  ObjectInputStream in = new
ObjectInputStream(socket.getInputStream())
        ) {
            Object inputObject = in.readObject();
            if (inputObject instanceof NetworkRequest) {
                NetworkRequest request = (NetworkRequest) inputObject;
                NetworkResponse response = handleRequest(request);
                out.writeObject(response);
                out.flush();
            }
        } catch (Exception e) {
            System.err.println("Помилка обробки клієнта: " + e.getMessage());
        }
        // ... закриття сокета ...
    }

    private NetworkResponse handleRequest(NetworkRequest req) {
        String command = req.getCommand();
        Object payload = req.getPayload();
```

```
        try {
            switch (command) {
                case "LOGIN":
                    String[] loginData = (String[]) payload;
                    User user = userService.login(loginData[0], loginData[1]);
                    return new NetworkResponse(user != null, user);

                case "GET_USER_TRACKS":
                    int userId = (int) payload;
                    List<Track> tracks = trackService.getAllTracks(userId);
                    return new NetworkResponse(true, tracks);

                case "ADD_TRACK":
                    // Логіка додавання треку
                    // ...
                    return new NetworkResponse(true, track);

                // ... обробка інших команд ...

                default:
                    return new NetworkResponse(false, "Невідома команда: " +
command);
            }
        } catch (Exception e) {
                    return new NetworkResponse(false, "Помилка сервера: " +
e.getMessage());
        }
    }
}
```

Висновок: Виконуючи цю лабораторну роботу, я ознайомився з такими видами взаємодії компонентів розподілених систем, як «Client-Server», «Peer-to-Peer» та «Service-Oriented Architecture» (SOA). Особливу увагу я приділив практичній реалізації клієнт-серверної архітектури, інтегрувавши її в проєкт музичного програвача. Цей підхід було обрано як архітектурне рішення для централізації зберігання даних та бізнес-логіки на стороні сервера, що дозволило відокремити інтерфейс користувача від безпосередньої роботи з базою даних.

Під час розробки мережевої взаємодії я зрозумів, наскільки ефективним є використання сокетів та серіалізації об'єктів (NetworkRequest/NetworkResponse) для обміну даними між віддаленими вузлами. Реалізація багатопотокового оброблення запитів через ClientHandler забезпечила стабільну роботу сервера з кількома клієнтами одночасно. Це зробило архітектуру застосунку більш структуровану та безпечною, оскільки клієнтська частина тепер взаємодіє з даними виключно через захищений фасад (MusicSystemFacade), приховуючи складність мережевих запитів від шару представлення.

Контрольні запитання

1. Що таке клієнт-серверна архітектура?

Клієнт-серверна архітектура — це модель розподілених додатків, де виділяється два види компонентів: клієнти (представляють додаток користувачеві, надсилають запити) і сервери (зберігають дані, обробляють запити та надсилають відповіді).

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

Сервіс-орієнтована архітектура (SOA) — це модульний підхід до розробки програмного забезпечення, заснований на використанні розподілених, слабко пов’язаних сервісів, які оснащені стандартизованими інтерфейсами для взаємодії. Історично вона з'явилася як альтернатива монолітній архітектурі для спрощення інтеграції та зменшення вартості переробки систем.

3. Якими принципами керується SOA?

Основними принципами SOA є використання розподілених сервісів, слабка пов'язаність (loose coupling) компонентів та використання стандартизованих інтерфейсів і протоколів для взаємодії.

4. Як між собою взаємодіють сервіси в SOA?

Сервіси в SOA взаємодіють між собою виключно за рахунок обміну повідомленнями, без створення спеціальних інтеграцій для прямого доступу до

даних (наприклад, до спільної бази даних). Зазвичай це реалізується через веб-служби, що використовують протоколи HTTP, SOAP або REST.

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

Згідно з принципами SOA, сервіси реєструються у спеціальних каталогах (реєстрах), де будь-яка команда розробників може знайти їх та отримати інформацію про те, як їх використовувати. Часто для обміну даними та маршрутизації запитів використовується централізований компонент — шина даних (Enterprise Service Bus).

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

- Переваги: Простота розгортання та оновлення (достатньо оновити серверну частину, щоб клієнти працювали з новою версією), централізоване зберігання та обробка даних. У випадку "товстого клієнта" перевагою є розвантаження сервера та можливість роботи офлайн.
- Недоліки: Залежність від сервера (якщо сервер недоступний, робота зупиняється або обмежується), високе навантаження на серверну частину у випадку "тонких клієнтів".

7. У чому полягають переваги та недоліки однорангової моделі взаємодії (P2P)?

- Переваги: Відсутність єдиної точки відмови завдяки децентралізації, висока стійкість мережі до збоїв, розподіл навантаження та ресурсів між усіма учасниками (рівноправність вузлів).
- Недоліки: Складність контролю даних через децентралізацію, проблеми з безпекою, складність синхронізації даних та зниження ефективності пошуку ресурсів зі збільшенням кількості вузлів.

8. Що таке мікросервісна архітектура?

Мікросервісна архітектура — це підхід до створення додатку як набору невеликих, автономних служб, кожна з яких виконується у своєму процесі, реалізує специфічні бізнес-можливості та може розгортатися незалежно. Це стиль розробки, що дозволяє легко розвивати високоавтоматизовані системи.

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

Служби в мікросервісній архітектурі зазвичай взаємодіють між собою за допомогою легковагих протоколів, таких як HTTP/HTTPS, WebSockets або AMQP (для асинхронного обміну повідомленнями).

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?

Ні, такий підхід зазвичай є просто реалізацією багатошарової архітектури (Layered Architecture) в рамках одного додатка (моноліту). SOA передбачає, що сервіси є розподіленими, слабко пов'язаними та взаємодіють через мережеві протоколи і повідомлення, а не просто як класи-сервіси всередині одного процесу пам'яті