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 №3

«Основи проектування розгортання»

Виконав:
студент групи ІА–32
Лось Ярослав

Київ 2025

Мета: Навчитися проєктувати діаграми розгортання та компонентів для системи що проєктується, а також розробляти діаграми взаємодії, а саме діаграми послідовностей, на основі сценаріїв зроблених в попередній лабораторній роботі.

Зміст

Теоретичні відомості

Діаграма розгортання (Deployment Diagram)

Діаграми розгортання представляють фізичне розташування системи, показуючи, на якому фізичному обладнанні запускається та чи інша складова програмного забезпечення. Головними елементами діаграми є вузли, пов'язані інформаційними шляхами. Вузол (node) – це те, що може містити програмне забезпечення. Вузли бувають двох типів. Пристрій (device) – це фізичне обладнання: комп'ютер або пристрій, пов'язаний із системою. Середовище виконання (execution environment) – це програмне забезпечення, яке саме може включати інше програмне забезпечення, наприклад операційну систему або процес-контейнер (наприклад, вебсервер).

Між вузлами можуть стояти зв'язки, які зазвичай зображують у вигляді прямої лінії. Як і на інших діаграмах, у зв'язків можуть бути атрибути множинності (для показання, наприклад, підключення 2х і більше клієнтів до одного сервера) і назва. У назві, як правило, міститься спосіб зв'язку між двома вузлами – це може бути назва протоколу (HTTP, IPC) або технологія, що використовується для забезпечення взаємодії вузлів (.NET Remoting, WCF).

Діаграма компонентів

Діаграма компонентів UML є представленням проєктованої системи, розбитої на окремі модулі [3]. Залежно від способу поділу на модулі розрізняють три види діаграм компонентів: • логічні; • фізичні; • виконувані. Коли використовують логічне розбиття на компоненти, то у такому разі проєктовану систему віртуально уявляють як набір самостійних, автономних модулів (компонентів), що взаємодіють між собою. 41 Коли на діаграмі представляють фізичне розбиття, то в такому разі на діаграмі компонентів показують компоненти та залежності між ними. Залежності показують, що класи в з одного компонента використовують класи з іншого компонента. Фізична модель використовується для розуміння які компоненти повинні бути зібрані в

інсталяційний пакет. Також така діаграма показує зміни в якому компоненті будуть впливати на інші компоненти.

Компоненти можуть поділятися за фізичними одиницями – окремі вузли розподіленої системи – набір комп'ютерів і серверів; на кожному з вузлів можуть бути встановлені різні виконувані компоненти. Такий вид діаграм компонентів застарів і зазвичай замість нього використовують діаграму розгортань.

Діаграма компонентів розробляється для таких цілей:

• візуалізації загальної структури вихідного коду програмної системи;

• специфікації виконуваного варіанта програмної системи;

• забезпечення багаторазового використання окремих фрагментів програмного коду;

• представлення концептуальної та фізичної схем баз даних.

Діаграми послідовностей

Діаграма послідовностей (Sequence Diagram) – це один із типів діаграм у моделюванні UML (Unified Modeling Language), який використовується для моделювання взаємодії між об'єктами системи у певній послідовності часу. Вона відображає, як об'єкти обмінюються повідомленнями, показуючи порядок і логіку виконання операцій.

Діаграма складається з таких основних елементів:

Актори (Actors): Зазвичай позначаються піктограмами або назвами. Це користувачі чи інші системи, які взаємодіють із системою. Актори можуть бути зовнішніми стосовно моделювання системи.

Об'єкти або класи: Розміщуються горизонтально на діаграмі. Вони позначаються прямокутниками з іменем об'єкта або класу під прямокутником. Кожен об'єкт має «життєвий цикл», який представлений вертикальною пунктирною лінією (лінія життя).

Повідомлення: Це лінії зі стрілками, які з'єднують об'єкти. Вони показують передачу повідомлень чи виклик методів. Стрілка може бути синхронною (звичайна стрілка) або асинхронною (лінія з відкритим трикутником) та з пунктирною лінією, що показує повернення результату.

Активності: Вказують періоди, протягом яких об'єкт виконує певну дію. На діаграмі це позначається прямокутником, накладеним на лінію життя.

Контрольні структури: Використовуються для відображення умов, циклів або альтернативних сценаріїв. Наприклад, блоки "alt" (альтернатива) або "loop" (цикл).

Основні кроки створення діаграми послідовностей:

- визначити акторів і об'єкти, які беруть участь у сценарії;
- побудувати їхні лінії життя; •

розробити послідовність передачі повідомлень між об'єктами;

- додати умовні блоки або цикли за необхідності.

Хід роботи

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.

1. Діаграма розгортання системи

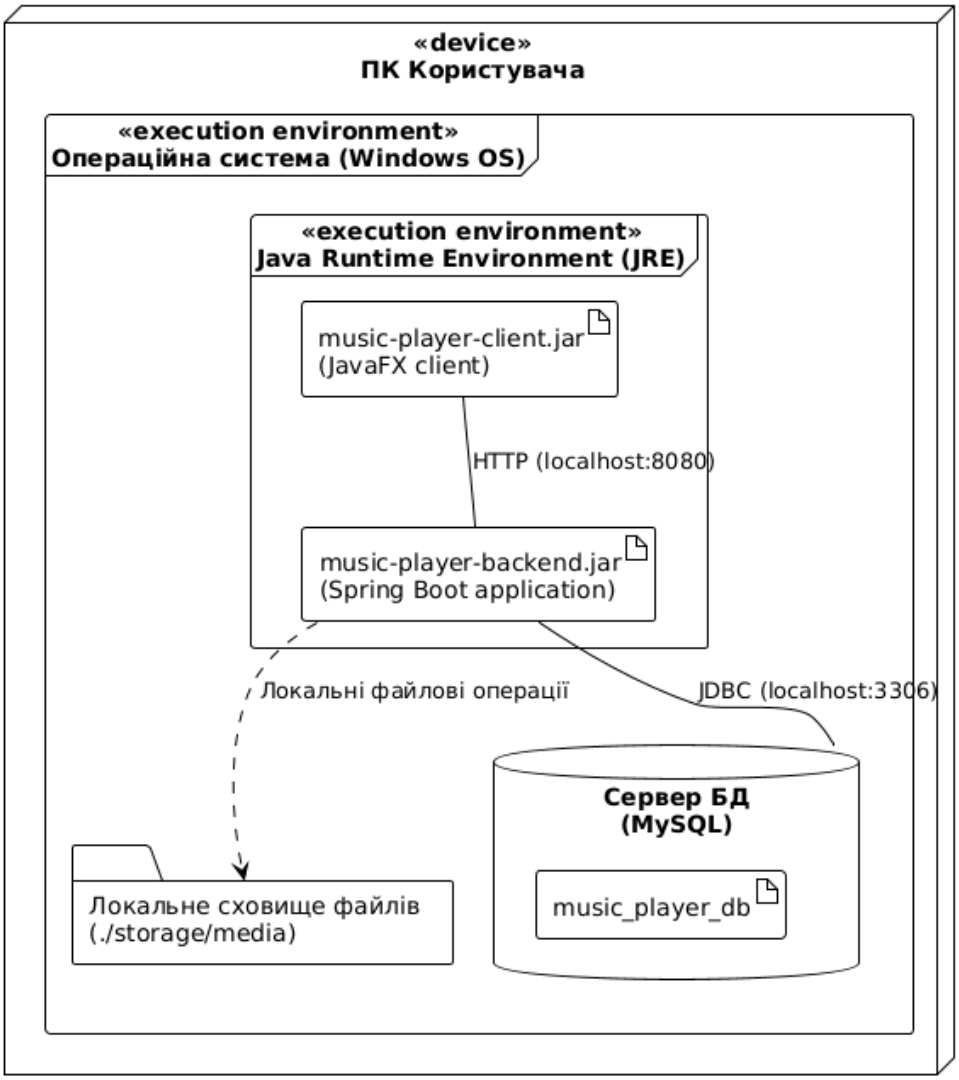

Рис.1 – Діаграма розгортання системи

Діаграма розгортання візуалізує фізичну архітектуру проєктованої системи. Вона демонструє, як програмні компоненти музичного плеєра розміщуються та взаємодіють між собою в рамках одного фізичного пристрою для локального використання.

Діаграма включає наступні ключові елементи:

1. Вузол «ПК Користувача» (<<device>>):

Це основний вузол, що представляє фізичний комп'ютер користувача. Він виступає як контейнер для операційної системи та всього програмного забезпечення, необхідного для роботи системи.

2. Середовища виконання (<<execution environment>>):

Діаграма показує два вкладених середовища виконання, що відображає ієрархію програмних шарів:

- Операційна система (Windows OS): Базове середовище, яке керує апаратними ресурсами комп'ютера та надає платформу для запуску інших програм.
- Java Runtime Environment (JRE): Спеціалізоване середовище, розгорнуте всередині ОС. Воно необхідне для виконання скомпільованого коду Java, представленого у вигляді артефактів .jar.

3. Артефакти (<<artifact>>):

Це скомпільовані, готові до розгортання частини системи, які запускаються в середовищі JRE. Система складається з двох основних артефактів, що реалізують клієнт-серверну логіку:

- **music-player-client.jar**: Клієнтська частина додатку, що відповідає за графічний інтерфейс користувача (GUI). Вона розроблена з використанням технології JavaFX.
- **music-player-backend.jar**: Серверна частина, реалізована на Spring Boot. Вона містить всю бізнес-логіку, API для взаємодії з клієнтом та логіку роботи з даними.

4. **Зовнішні ресурси та сховища даних:**

Це компоненти, які працюють поза JRE, але в межах того ж фізичного пристрою, і використовуються бекендом для зберігання даних:

- Сервер БД (MySQL): Система управління базами даних, що відповідає за зберігання структурованої інформації: дані користувачів, плейлисти, метадані треків. Бекенд взаємодіє із конкретною базою даних music_player_db.
- Локальне сховище файлів (./storage/media): Директорія у файловій системі комп'ютера, яка використовується для зберігання самих музичних файлів (у форматах MP3, FLAC).

Опис взаємозв'язків:

- Клієнт - Бекенд: Взаємодія між music-player-client.jar та music-player-backend.jar здійснюється за протоколом HTTP. Клієнт надсилає запити на локальний сервер (localhost:8080) для отримання даних або виконання дій.
- Бекенд - Сервер БД: Серверна частина звертається до бази даних MySQL через JDBC (Java Database Connectivity) на стандартний порт 3306. Цей зв'язок використовується для всіх CRUD-операцій зі структурованими даними.
- Бекенд - Локальне сховище файлів: Для доступу до музичних файлів бекенд виконує локальні файлові операції, використовуючи стандартні бібліотеки вводу-виводу Java.

2. Діаграма компонентів

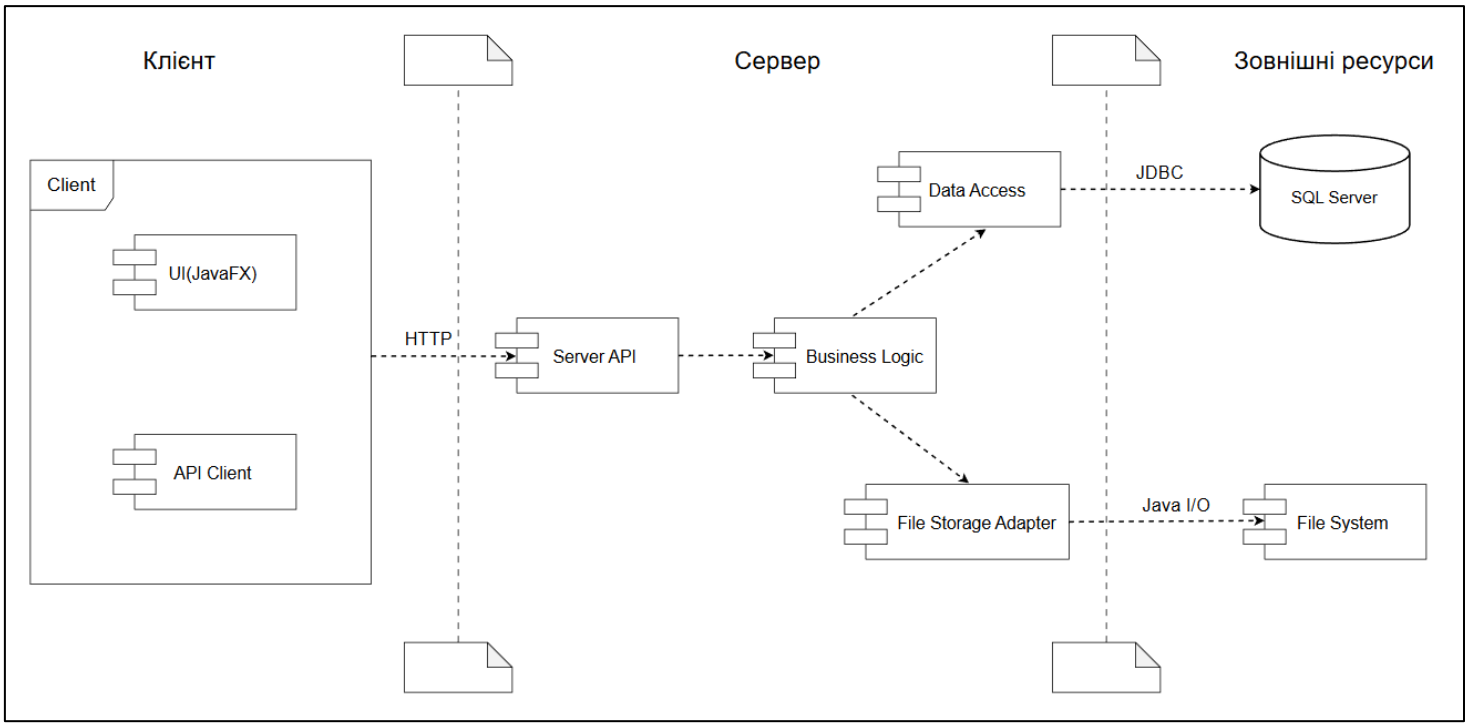

Рис. 2 - Діаграма компонентів

Діаграма компонентів представляє логічну архітектуру системи, структуровану у вигляді трьох основних шарів: Клієнт, Сервер та Зовнішні ресурси. Вона візуалізує модульний склад програми, залежності між ключовими компонентами та чіткий потік даних, що є основою для реалізації клієнт-серверної архітектури.

1. Шар «Клієнт»

Цей шар об'єднує всі компоненти, відповідальні за користувацький інтерфейс та взаємодію з користувачем. Він є "обличчям" системи, але не містить основної бізнес-логіки.

- UI (JavaFX): Основний компонент, що реалізує графічний інтерфейс користувача (GUI). Він відповідає за відображення списків треків, плейлистів, елементів керування відтворенням та приймає дії користувача.
- API Client: Інкапсулює всю логіку комунікації з серверною частиною. Цей компонент формує та відправляє HTTP-запити на серверний API для отримання даних або виконання дій. Він забезпечує чіткий інтерфейс для взаємодії з сервером.

2. Шар «Сервер»

Цей шар є ядром системи, що містить всю бізнес-логіку. Він працює у фоновому режимі, не має власного UI та обробляє всі запити від клієнта.

- Server API: Вхідна точка сервера. Цей компонент приймає HTTP-запити від API Client, аналізує їх і виступає в ролі шлюзу, делегуючи подальшу обробку відповідним сервісам.
- Business Logic (Service): Центральний компонент, що керує виконанням всіх операцій. Він містить основну логіку системи: керування користувачами, плейлистами, обробку відтворення та історії прослуховувань.
- Data Access (Repository): Надає абстрактний шар доступу до бази даних. Його компоненти приховують деталі виконання SQL-запитів, надаючи сервісному шару прості методи для роботи з даними.

- File Storage Adapter: Ізолює та інкапсулює логіку роботи з файловою системою. Він надає інтерфейс для модуля Business Logic для доступу до музичних файлів, приховуючи низькорівневі деталі.

3. Шар «Зовнішні ресурси»

Представляє зовнішні ресурси, з якими взаємодіє сервер для зберігання та отримання даних.

- SQL Server: Зовнішнє сховище даних. Модуль Data Access звертається до нього через протокол JDBC для збереження та отримання структурованої інформації: профілів користувачів, плейлистів, метаданих треків та історії прослуховувань.
- File System: Файлова система комп'ютера. Модуль File Storage Adapter взаємодіє з нею через стандартну бібліотеку Java I/O для зберігання та читання самих музичних файлів.

3. Діаграми послідовностей

Назва: **Слухати потокову музику**

Короткий опис: сценарій описує складний процес взаємодії між компонентами системи для відтворення треку, що зберігається на сервері.

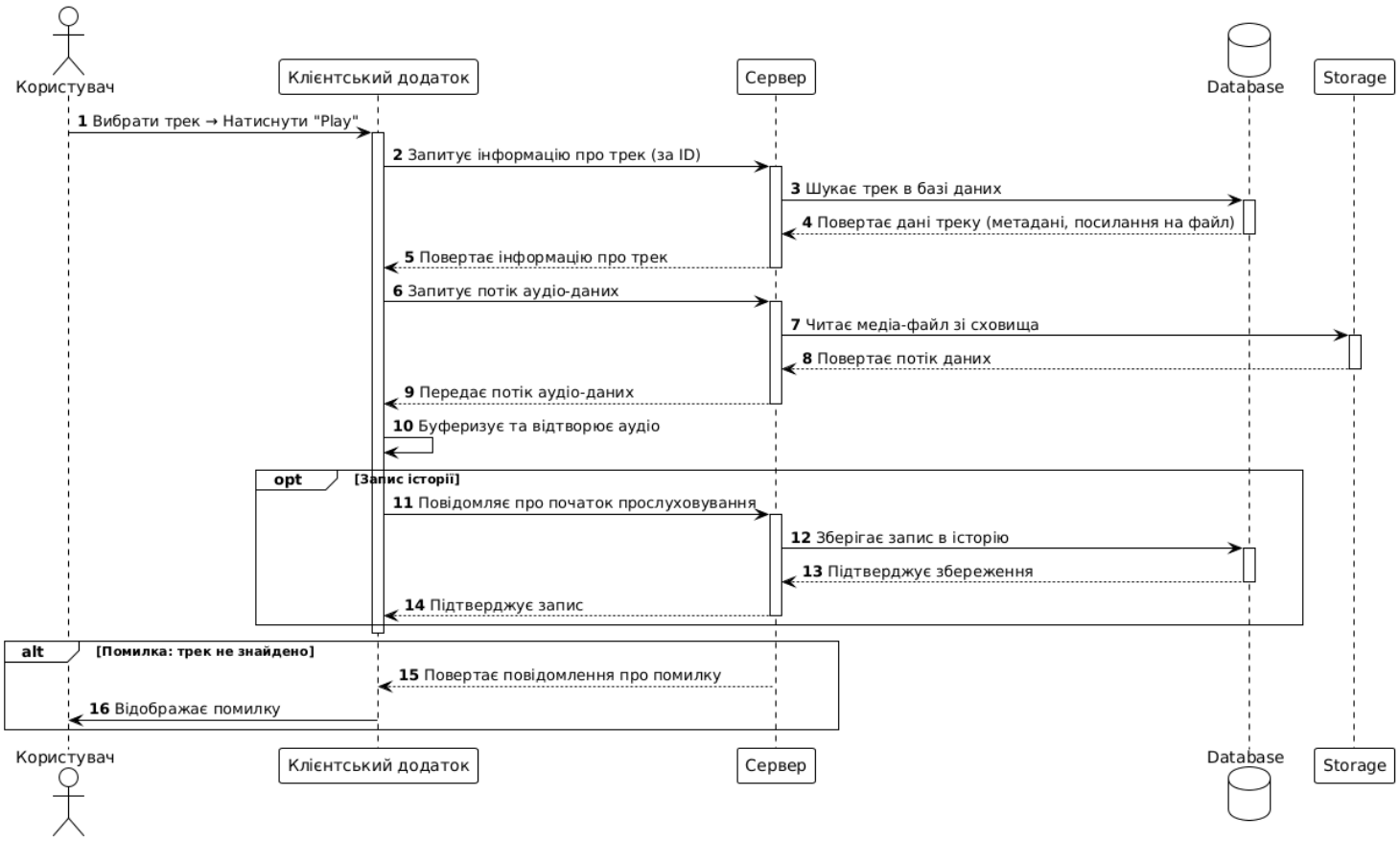

Рис. 3 - Діаграма послідовностей для прослуховування потокової музики

Детальний опис потоку подій:

1. Ініціація: користувач у клієнтському додатку обирає потрібний трек та натискає "Play".
2. Запит інформації: клієнтський додаток відправляє на сервер запит з проханням надати інформацію про обраний трек (за його ідентифікатором).
3. Пошук даних: сервер, отримавши запит, звертається до бази даних для пошуку запису про цей трек.
4. Отримання даних: база даних повертає серверу необхідні дані, такі як метадані та внутрішнє посилання на розташування аудіофайлу.
5. Відповідь клієнту: сервер відправляє отриману інформацію про трек назад до клієнтського додатку.
6. Запит на аудіо: клієнтський додаток, отримавши інформацію, робить другий запит до сервера, цього разу безпосередньо за потоком аудіо-даних.
7. Доступ до файлу: сервер звертається до сховища файлів, для читання медіа-файлу.

8. Передача потоку: сховище починає віддавати потік даних Серверу.
9. Стрімінг: сервер транслює отриманий потік аудіо-даних до клієнтського додатку.
10. Відтворення: клієнтський додаток буферизує отримані дані та починає відтворення музики для Користувача.

Опціональний потік: Запис історії прослуховування
1. Повідомлення про подію: після початку відтворення клієнтський додаток відправляє на сервер окреме повідомлення про початок прослуховування.
2. Збереження в БД: сервер обробляє це повідомлення та зберігає відповідний запис в історію прослуховувань у базу даних.
3. Підтвердження збереження: БД повідомляє сервер про успішне збереження.
4. Фінальне підтвердження: сервер відправляє клієнтському додатку підтвердження, що подія була успішно записана.

Альтернативний потік: Помилка (трек не знайдено)
1. Повідомлення про помилку: якщо на кроці 4 БД не знаходить запис про трек, сервер перериває процес і відправляє клієнтському додатку повідомлення про помилку.
2. Відображення помилки: клієнтський додаток отримує повідомлення та відображає його користувачеві.

Назва: **Створення нового плейлиста**
Короткий опис: Сценарій описує процес, за допомогою якого зареєстрований користувач створює новий персональний плейлист у системі.

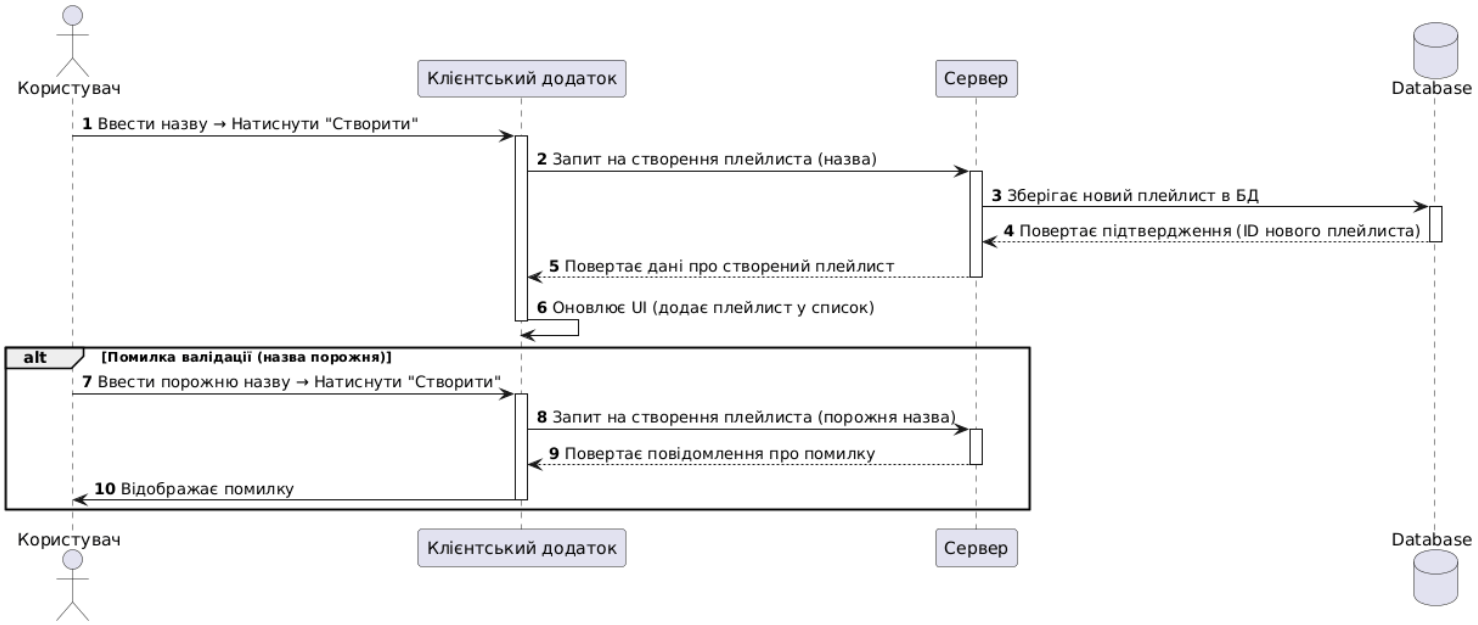

Рис. 4 – Діаграма послідовностей для створення нового плейлиста

Детальний опис потоку подій (Основний потік):

1. Ініціація: Користувач вводить назву для нового плейлиста у відповідне поле в клієнтському додатку та натискає кнопку "Створити".
2. Запит на створення: Клієнтський додаток відправляє на сервер HTTP-запит, що містить назву майбутнього плейлиста та ідентифікатор користувача.
3. Збереження в БД: Сервер, отримавши запит, створює новий об'єкт плейлиста та передає команду на збереження цього запису в базу даних.
4. Підтвердження від БД: База даних зберігає новий запис та повертає серверу підтвердження, зазвичай разом з унікальним ідентифікатором (ID) щойно створеного плейлиста.
5. Відповідь клієнту: Сервер формує успішну відповідь, що містить дані про новий плейлист (ID, назву), та відправляє її клієнтському додатку.
6. Оновлення інтерфейсу: Клієнтський додаток, отримавши підтвердження, оновлює свій стан та відображає новий плейлист у списку плейлистів користувача.

Альтернативний потік: Помилка валідації

1. Повідомлення про помилку: Якщо на кроці 3 сервер визначає, що передана назва плейлиста є некоректною (наприклад, порожньою або занадто довгою),

він перериває процес, не звертаючись до бази даних, і відправляє клієнтському додатку повідомлення про помилку валідації.

2. Відображення помилки: Клієнтський додаток отримує повідомлення та відображає його користувачеві (наприклад, "Назва не може бути порожньою").

Висновки

У ході виконання лабораторної роботи було створено основні архітектурні діаграми системи, що дало змогу детально відобразити як її статичну структуру, так і динамічну поведінку. Розроблена діаграма розгортання відобразила фізичну архітектуру системи, показавши реалізацію клієнт-серверного застосунку у вигляді єдиного артефакту, розміщеного на комп’ютері користувача. Деталізована діаграма компонентів дала змогу визначити модульну структуру програми, логічні рівні та залежності між ними, а також продемонструвала компоненти, що реалізують ключові шаблони проєктування.

Для аналізу динамічної взаємодії було побудовано дві діаграми послідовностей.

Таким чином, отримано узгоджений і повний набір UML-діаграм, що формує архітектурну основу для подальшої розробки та впровадження функціоналу системи

Контрольні запитання

1. Що собою становить діаграма розгортання?

Діаграма розгортання (deployment diagram) показує фізичну (апаратно-програмну) архітектуру системи: які вузли (сервери, пристрої, контейнери, середовища виконання) існують, які артефакти (виконувані файли, бібліотеки, контейнери) на них розгорнуті та як вузли з'єднані між собою (мережа, канали зв'язку). Призначена для планування інфраструктури і розгортання системи в реальному середовищі.

2. Які бувають види вузлів на діаграмі розгортання?

- Device (пристрій) — фізичний хост (сервер, робоча станція, мобільний телефон).
- ExecutionEnvironment (середовище виконання) — логічне/віртуальне середовище (JVM, контейнер Docker, OS process).
- Node (загальний вузол) — узагальнений тип вузла (може мати стереотипи). Також розрізняють фізичні вузли та віртуальні/логічні (контейнери, віртуальні машини, кластери).

3. Які бувають зв'язки на діаграмі розгортання?

- Communication path — шлях зв'язку між вузлами (мережеве з'єднання).
- Deployment (залежність розгортання) — відношення «артефакт deploy'иться на вузол» (позначається <<deploy>>).
- Association / Dependency — загальні асоціації чи залежності між вузлами.
- Generalization — ієрархія вузлів (рідше використовується).

4. Які елементи присутні на діаграмі компонентів?

- Component (компонент) — модуль програми (фізичний або логічний).
- Interface (інтерфейс) — надані (provided, «лолі-п»/круг) і потрібні (required, «гніздо») інтерфейси.
- Port (порт) — точка взаємодії компонента.

- Connector / Assembly connector — з’єднання між компонентами/портами.
- Artifact, Package, Note — допоміжні елементи для організації та документування.

5. Що становлять собою зв’язки на діаграмі компонентів?

Зв'язки на діаграмі компонентів переважно представляють залежність (Dependency). Такий зв'язок показує, що один компонент (клієнт) для своєї роботи використовує класи або інтерфейси, надані іншим компонентом (постачальником). По суті, це означає, що зміни в компоненті-постачальнику можуть вплинути на роботу компонента-клієнта.

6. Які бувають види діаграм взаємодії?

- Sequence diagram (діаграма послідовностей)
- Communication (collaboration) diagram (діаграма комунікації)
- Interaction Overview diagram
- Timing diagram (таймінг-діаграма)

7. Для чого призначена діаграма послідовностей?

Діаграма послідовностей моделює часовий порядок повідомлень між об’єктами/акторами для конкретного сценарію (варіанту використання): показує хто з ким спілкується, в які моменти відбуваються виклики, активності, відповіді, альтернативні та повторювані потоки. Використовується для проєктування поведінки та уточнення сценаріїв реалізації.

8. Які ключові елементи можуть бути на діаграмі послідовностей?

Ключовими елементами діаграми послідовностей є:

1. Актор (Actor): Ініціатор сценарію (користувач або зовнішня система).
2. Об'єкт (Object): Учасник взаємодії, від якого йде лінія життя (lifeline).
3. Повідомлення (Message): Стрілка між лініями життя, що позначає виклик методу або передачу даних.

4. Активація (Activation): Прямокутник на лінії життя, що показує час виконання операції.

5. Фрагменти взаємодії: Спеціальні блоки для моделювання логіки, такі як alt (альтернатива), opt (опція) та loop (цикл).

9. Як діаграми послідовностей пов'язані з діаграмами варіантів використання?

Кожна діаграма послідовностей зазвичай **реалізує (деталізує)** конкретний варіант використання (use case) або його потік подій: use case описує що має статися (сценарій), а sequence diagram — як саме це відбувається між об'єктами в часовому вимірі (основний та альтернативні потоки).

10. Як діаграми послідовностей пов'язані з діаграмами класів?

Sequence diagram показує повідомлення між екземплярами (objects) — ці екземпляри належать до класів, визначених на діаграмі класів. Зі sequence diagram часто витікають: які методи потрібні класам, які атрибути/зв'язки використовуються, та як класи взаємодіють — тобто sequence допомагає уточнити і доповнити модель класів та сигнатури методів.

Додаток А

```
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.List;

public class MainApp extends Application {
codeCode
private final HttpClient httpClient = HttpClient.newHttpClient();
private final ObjectMapper objectMapper = new ObjectMapper();
private final String serverUrl = "http://localhost:8080/api/playlists";

private ListView<Playlist> playlistListView = new ListView<>();
private ObservableList<Playlist> playlists = FXCollections.observableArrayList();

public static void main(String[] args) {
    launch(args);
}

@Override
public void start(Stage primaryStage) {
```

```
    primaryStage.setTitle("Музичний програвач");

    Button createButton = new Button("Створити новий плейлист");
    createButton.setOnAction(e -> handleCreatePlaylist());

    Button refreshButton = new Button("Оновити список");
    refreshButton.setOnAction(e -> loadPlaylists());

    playlistListView.setItems(playlists);

    VBox layout = new VBox(10);
    layout.setPadding(new Insets(10));
    layout.getChildren().addAll(new Label("Мої плейлисти:"), playlistListView,
createButton, refreshButton);

    Scene scene = new Scene(layout, 400, 300);
    primaryStage.setScene(scene);
    primaryStage.show();

    loadPlaylists();
}

private void loadPlaylists() {
    HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(serverUrl))
            .GET()
            .build();
    try {
        HttpResponse<String> response = httpClient.send(request,
HttpResponse.BodyHandlers.ofString());
```

```
        List<Playlist> serverPlaylists = objectMapper.readValue(response.body(), new
TypeReference<>() {});
        playlists.setAll(serverPlaylists);
    } catch (Exception e) {
        showAlert("Помилка завантаження", "Не вдалося завантажити плейлисти з
сервера.");
        e.printStackTrace();
    }
}

private void handleCreatePlaylist() {
    String name = CreatePlaylistForm.display();

    if (name != null && !name.trim().isEmpty()) {
        String json = String.format("{\"name\":\"%s\"}", name);

        HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(serverUrl))
                .header("Content-Type", "application/json")
                .POST(HttpRequest.BodyPublishers.ofString(json))
                .build();
        try {
            HttpResponse<String> response = httpClient.send(request,
HttpResponse.BodyHandlers.ofString());
            if (response.statusCode() == 200) {
                loadPlaylists();
            } else {
                showAlert("Помилка", "Не вдалося створити плейлист. Статус: " +
response.statusCode());
            }
        } catch (Exception e) {
```

```
            showAlert("Помилка з'єднання", "Не вдалося відправити дані на
сервер.");
            e.printStackTrace();
        }
    }
}

private void showAlert(String title, String message) {
    Alert alert = new Alert(Alert.AlertType.ERROR);
    alert.setTitle(title);
    alert.setHeaderText(null);
    alert.setContentText(message);
    alert.showAndWait();
}
}


import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.VBox;
import javafx.stage.Modality;
import javafx.stage.Stage;

public class CreatePlaylistForm {
    private static String playlistName = null;

    public static String display() {
        Stage window = new Stage();
        window.initModality(Modality.APPLICATION_MODAL);
        window.setTitle("Створити новий плейлист");
```

```
        window.setMinWidth(300);

        Label label = new Label("Введіть назву плейлиста:");
        TextField nameInput = new TextField();
        Button saveButton = new Button("Зберегти");

        saveButton.setOnAction(e -> {
            playlistName = nameInput.getText();
            window.close();
        });

        VBox layout = new VBox(10);
        layout.setPadding(new Insets(10));
        layout.getChildren().addAll(label, nameInput, saveButton);

        Scene scene = new Scene(layout);
        window.setScene(scene);
        window.showAndWait();

        return playlistName;
    }
}


public class Playlist {
    private Long id;
    private String name;
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
```

```
    @Override
    public String toString() {
        return name;
    }
}
```