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 лабораторної роботи: «Взаємодія компонентів системи»

Тема проєкта: «Аудіо редактор»

Виконала:
студентка групи ІА-33
Котик Іванна

Перевірив:
асистент кафедри ІСТ
Мягкий Михайло Юрійович

Київ 2025

Зміст

Тема: Взаємодія компонентів системи.

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

Тема проєкта: 5. Аудіо редактор (singleton, adapter, observer, mediator, composite, client server). Аудіо редактор повинен володіти наступним функціоналом: представлення аудіо даних будь-якого формату в WAVE-формі, вибір і подальші операції копіювання / вставки / вирізання / деформації по сегменту аудіозапису, можливість роботи з декількома звуковими доріжками, кодування в найбільш поширених форматах (ogg, flac, mp3).

Короткі теоретичні відомості:

Клієнт-серверна архітектура

Клієнт-серверні додатки являють собою найпростіший варіант розподілених додатків, де виділяється два види додатків: клієнти (представляють додаток користувачеві) і сервери (використовується для зберігання і обробки даних). Розрізняють тонкі клієнти і товсті клієнти.

Тонкий клієнт – клієнт, який повністю всі операції (або більшість, пов'язаних з логікою роботи програми) передає для обробки на сервер, а сам зберігає лише візуальне уявлення одержуваних від сервера відповідей. Грубо кажучи, тонкий клієнт – набір форм відображення і канал зв'язку з сервером. Прикладом тонкого клієнта є класичні Web-застосунки.

У такому варіанті використання майже все навантаження лягає на сервер або групу серверів.

Перевагою таких моделей є простота розгортання, тому що оновлювати потрібно лише сервери і в результаті клієнти з наступними запитами автоматично будуть працювати з оновленою системою.

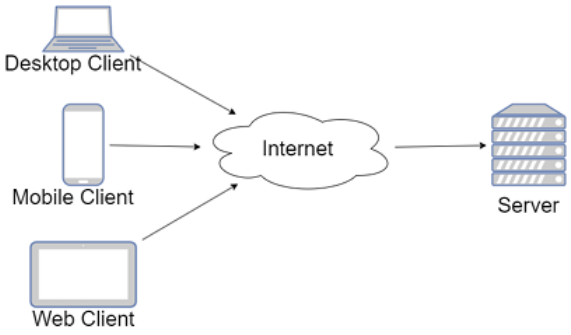

Рисунок 1 – Клієнт-серверна архітектура

Товстий клієнт – антипод тонкого клієнта, більшість логіки обробки даних містить на стороні клієнта. Це сильно розвантажує сервер. Сервер в таких випадках зазвичай працює лише як точка доступу до деякого іншого ресурсу (наприклад, бази даних) або сполучна ланка з іншими клієнтськими комп'ютерами. Перевагою такого підходу є менші вимоги до серверної частини. Також перевагою, при певному підході до реалізації, є можливість працювати клієнтам без тимчасового доступу до серверу. Прикладом товстого клієнта можна назвати мобільні застосунки, або десктоп застосунки. Наприклад, Evernote, Viber, MS Outlook, комп’ютерні антивіруси, ігри, що потребують інсталяції (The Sims, GTA, …) та інші.

Проміжним варіантом можна назвати SPA (Single Page Application) – це товсті Web-клієнти, які при старті кожен раз завантажуються з сервера, а надалі працюють з сервером через web-API. З одного боку більшу частину логіки вони відпрацьовують на клієнтський стороні, за рахунок чого зменшується серверне навантаження. Також оновлення простіше ніж для товстих клієнтів. Але такі застосунки не працюють, якщо сервер не доступний.

Клієнт-серверна взаємодія, як правило, організовується за допомогою 3-х рівневої структури: клієнтська частина, загальна частина, серверна частина.

Оскільки велика частина даних загальна (класи, використовувані системою), їх прийнято виносити в загальну частину (middleware) системи.

Клієнтська частина містить візуальне відображення і логіку обробки дії користувача; код для встановлення сеансу зв'язку з сервером і виконання відповідних викликів.

Серверна частина містить основну логіку роботи програми (бізнес-логіку) або ту її частину, яка відповідає зберіганню або обміну даними між клієнтом і сервером або клієнтами.

Peer-to-Peer архітектура

Peer-to-Peer (P2P) архітектура – це модель мережевої взаємодії, в якій кожен вузол (комп'ютер або пристрій) є одночасно клієнтом і сервером. У цій архітектурі всі вузли мають рівні права та можливості для обміну даними, ресурсами або виконання завдань. На відміну від клієнт-серверної моделі, де є чітке розділення на клієнти й сервери, P2P-мережа дозволяє учасникам взаємодіяти безпосередньо, без необхідності в централізованому сервері.

Основними принципами P2P-архітектури є:

- Децентралізація – відсутність центрального сервера, що зменшує залежність від одного вузла, підвищуючи стійкість мережі до збоїв і атак.
- Рівноправність вузлів – кожен вузол може виконувати одночасно функції клієнта (отримувати ресурси) і сервера (надавати ресурси).
- Розподіл ресурсів – вузли надають доступ до своїх власних ресурсів, таких як обчислювальна потужність, дисковий простір або файли.

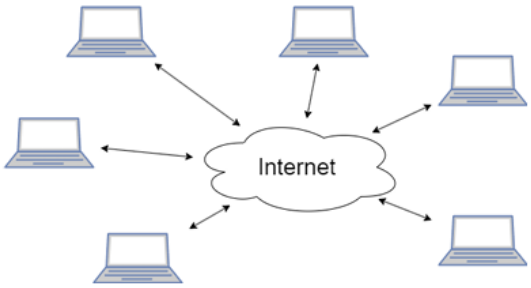

Рисунок 2 – Peer-to-Peer архітектура

Основними сферами де peer-to-peer архітектура знайшла широке застосування є файлообмінники (BitTorrent), криптовалюти та інші блокчейн технології, інтернет телефонія та відеоконференції (Skype, Zoom), розподілені обчислення (SETI@home, BOINC).

До основних проблемних зон можна віднести безпеку, синхронізацію даних та пошук ресурсів. Через централізацію складно контролювати дані, які передаються. Ефективність пошуку даних знижується зі збільшенням кількості вузлів у мережі і для підвищеня ефективності пошуку потрібно застосовувати спеціальні алгоритми.

Сервіс-орієнтована архітектура

Сервіс-орієнтована архітектура (SOA, англ. service-oriented architecture) – модульний підхід до розробки програмного забезпечення, заснований на використанні розподілених, слабо пов’язаних (англ. Loose coupling) сервісів або служб, оснащених стандартизованими інтерфейсами для взаємодії за стандартизованими протоколами.

Історично сервіс-орієнтована архітектура появилася як альтернатива монолітній архітектурі, в які вся система розроблялася та розгорталася як одне ціле.

Програмні комплекси, розроблені відповідно до сервіс-орієнтованою архітектурою, зазвичай реалізуються як набір веб-служб (або веб-сервісів), які, як правило, взаємодіють по HTTP з використанням SOAP або REST. Ці служби надають певні бізнес-функції, наприклад, отримання інформації про наявність матеріалів на складі.

Сервіси взаємодіють між собою тільки за рахунок обміну повідомленнями, без створення спеціальних інтеграцій для доступу до однієї інформації, наприклад, до однієї бази даних.

Сервіси також можуть бути реалізовані як обгортки навколо застарілої системи. Це робиться для зменшення вартості переробки системи, а також спрощення інтеграції існуючих монолітних систем в нову архітектуру.

Згідно SOA сервіси реєструються на спеціальних сервісах і будь-яка команда розробників, якій потрібен доступ може знайти їх та використовувати.

Часто реалізація SOA покладається на використання централізованого програмного компонента для обміну даними – шину даних (Enterprise Service

Bus).

Мікросервісна архітектура є подальшим розвитком сервіс-орієнтованої архітектури з використанням нових напрацювань у інформаційних технологіях.

Мікро-сервісна архітектура.

Сама назва дає зрозуміти, що мікро-сервісна архітектура є підходом до створення серверного додатку як набору малих служб. Це означає, що архітектура мікро-сервісів головним чином орієнтована на серверну частину, не дивлячись на те, що цей підхід так само використовується для зовнішнього інтерфейсу, де кожна служба виконується в своєму процесі і взаємодіє з іншими службами за такими протоколами, як HTTP/HTTPS, WebSockets чи AMQP. Кожен мікросервіс реалізує специфічні можливості в предметній області і свою бізнес-логіку в

рамках конкретного обмеженого контексту, повинна розроблятись автономно і розвертатись незалежно.

Визначення мікросервісів із книги Іраклі Надарейшвілі, Ронні Мітра, Метта Маклярті та Майка Амундсена (О'Рейлі) «Архітектура мікросервісів»:

«Мікросервіс – це компонент із чітко визначеними межами, який можна розгортати незалежно, і підтримує взаємодію за допомогою зв'язку на основі повідомлень. Архітектура мікросервісів – це стиль розробки високоавтоматизованих систем програмного забезпечення, що легко розвивати та яке складається з мікросервісів, орієнтованих на певні можливості».

Мікросервіси забезпечують чудові можливості супроводження в величезних комплексних системах з високою масштабуємістю за рахунок створення додатків, заснованих на множині незалежно розгортуючих служб з автономними життєвими циклами.

Хід роботи:

1. Постановка задачі.

У межах лабораторної роботи необхідно створити клієнт-серверну програмну систему, що включає клієнтську частину, серверну частину та проміжний модуль (middleware). Клієнтський застосунок відповідає за відображення даних, взаємодію з користувачем та виконання локальної логіки редагування аудіо. Серверна частина забезпечує оброблення запитів, реалізацію бізнес-логіки, доступ до бази даних і збереження інформації про користувачів, проєкти, доріжки та сегменти. Middleware-модуль визначає єдиний формат обміну даними між компонентами системи та включає спільні DTO-класи.

Методичні вказівки дозволяють організувати зв'язок між клієнтом і сервером за допомогою WCF, TcpClient або .NET Remoting. Однак у даному проєкті було свідомо обрано REST-підхід, що реалізується через HTTP/HTTPS. Це рішення зумовлене тим, що REST не має обмежень щодо операційної системи, середовища виконання або мови програмування, що забезпечує повну технологічну нейтральність архітектури. На відміну від WCF та .NET Remoting, які історично орієнтовані на Windows та .NET Framework, REST природно інтегрується з

широким спектром сучасних платформ, що дозволяє поєднати клієнтську частину, реалізовану на JavaFX, та серверну частину на Spring Boot, зберігаючи цілісність архітектури та дотримуючись єдиних принципів побудови API. Такий підхід відповідає сучасним промисловим стандартам і забезпечує можливість масштабування, тестування, документування та подальшої інтеграції з іншими сервісами незалежно від їхньої внутрішньої технологічної реалізації.

REST забезпечує передавання структурованих даних у форматі JSON, який є універсальним для будь-якого середовища, легко серіалізується та десеріалізується, підтримує версіонування та є загальноприйнятим у сучасних API. Це дає змогу зберегти чітку багаторівневу архітектуру системи: контролери відповідають за оброблення HTTP-запитів, сервіси — за бізнес-логіку, репозиторії — за доступ до даних, а DTO — за стандартизований обмін інформацією між клієнтом і сервером.

Таким чином, у межах даної роботи необхідно реалізувати клієнтську частину, серверну частину та спільний middleware-модуль, організувати взаємодію між ними через REST API, а також підготувати UML-діаграму класів та фрагменти коду, що демонструють структуру, взаємодію та принципи побудови створеної клієнт-серверної архітектури.

2. Опис архітектури клієнт–серверної системи.

Архітектура створеної системи ґрунтується на класичній моделі клієнт–серверної взаємодії, у межах якої клієнтський застосунок відповідає за відображення інформації, взаємодію з користувачем та виконання локальних операцій редагування аудіо, тоді як серверна частина реалізує централізоване управління даними, виконує бізнес-логіку, забезпечує доступ до бази даних і відповідає за збереження та відновлення всієї структури аудіопроєктів. У ролі проміжного компонента виступає спільний модуль (middleware), який містить

DTO-класи та визначає узгоджений контракт обміну даними між клієнтом і сервером, гарантує їх технологічну сумісність і чітку стандартизацію форматів запитів та відповідей.

У даній системі клієнтський застосунок виступає у ролі «товстого клієнта» (fat client), оскільки містить значну частину прикладної логіки, включаючи рендеринг інтерфейсу, локальну обробку аудіо, виконання операцій редагування та координацію внутрішніх модулів. Сервер, у свою чергу, реалізує переважно зберігання даних та бізнес-логіку, а не функції оброблення мультимедіа чи UI, що є характерним для fat-client архітектури.

Клієнтський застосунок побудований на JavaFX і організований як структурований модульний проєкт, що використовує патерни Composite, Mediator та Observer. За допомогою Composite реалізовано модель аудіопроєкту, у якій проєкт містить набір треків, а кожен трек складається з окремих сегментів. Така ієрархічна структура дозволяє працювати з аудіо так само, як зі складеними об’єктами, що значно полегшує виконання операцій редагування. Патерн Mediator забезпечує централізовану координацію роботи модулів клієнта, дозволяючи їм взаємодіяти не безпосередньо, а через єдиний керівний елемент, що суттєво знижує зв’язність системи. Додатково використана подієва модель, яка дозволяє модулю AudioEditor повідомляти інші компоненти про події відтворення, зміну курсора або оновлення аудіосегмента, завдяки чому взаємодія між логічними частинами клієнта залишається динамічною та узгодженою. За роботу з мережею на клієнті відповідає ApiClient, який надсилає HTTP-запити на сервер і виконує серіалізацію та десеріалізацію об’єктів.

Middleware-модуль включає набір DTO, що визначають універсальний формат передавання інформації між клієнтом і сервером. Використання таких об’єктів дозволяє уніфікувати транспортні структури та відокремити внутрішні серверні сутності, пов’язані з базою даних, від зовнішнього API, який споживається клієнтом. DTO гарантують стабільність взаємодії, незалежно від того, як змінюється внутрішня логіка сервера або структура БД.

Серверну частину системи реалізовано у вигляді Spring Boot застосунку, структурованого за багатошаровою архітектурою. Контролери приймають HTTP-запити від клієнта та повертають результати в стандартизованому JSON-форматі. Уся бізнес-логіка зосереджена в сервісному рівні, який відповідає за створення проєктів, редагування треків, оброблення аудіосегментів, керування експортом та інші функції. Доступ до бази даних здійснюється через репозиторії Spring Data JPA, що дозволяє зберігати дані у вигляді сутностей, повністю інтегрованих із ORM-механізмами. Для перетворення сутностей у DTO застосовуються окремі мапери, що забезпечують поділ внутрішньої структури даних і транспортного рівня. Такий підхід дозволяє зберегти чистоту архітектури та спрощує подальший розвиток системи.

Обмін даними між клієнтом і сервером здійснюється через REST API, який підтримує чітко визначений набір операцій для роботи з проєктами, треками, сегментами та експортованими файлами. Використання формату JSON забезпечує універсальність та читабельність переданих структур, а HTTP/HTTPS гарантує сумісність між різними платформами й можливість безперешкодного використання Java-клієнта та Spring-сервера в межах однієї системи. Така взаємодія дозволяє клієнту отримувати актуальні дані з сервера, синхронізувати локальний стан проєкту з інформацією, що зберігається у базі, та працювати з аудіо у реальному часі.

Загальна архітектура системи відзначається модульністю, низькою зв’язністю та чітким розподілом відповідальностей. Поєднання патернів проєктування, багатошарової серверної структури та REST-взаємодії забезпечує надійну, зручну в підтримці та масштабовану клієнт–серверну платформу, яка здатна розвиватися та інтегруватися з іншими сервісами без змін у фундаментальних механізмах. Такий підхід гарантує стабільність, розширюваність і відповідність сучасним інженерним стандартам розробки прикладних програмних систем.

3. Діаграми, що описують стркутуру застосунку.

У цьому розділі подано UML-діаграми, які ілюструють структурну організацію клієнтсько-серверної системи AudioEditor. Діаграми демонструють будову основних модулів застосунку, внутрішні класи, їх взаємозв’язки та принципи взаємодії між компонентами.

Система складається з трьох головних частин: клієнтського застосунку, серверної частини та спільного модуля (middleware). Клієнтська частина включає модулі користувацького інтерфейсу, локальної логіки, патернів Composite та Mediator, а також мережевий модуль, що відповідає за REST-взаємодію з сервером. Серверна частина побудована за багатошаровою архітектурою й містить контролери, сервіси, репозиторії та JPA-сутності, що забезпечують обробку запитів і роботу з базою даних. Спільний модуль містить DTO-класи, які визначають уніфікований формат обміну даними між клієнтом і сервером.

Представлені нижче діаграми відображають структуру кожного з цих компонентів та показують цілісну архітектуру застосунку.

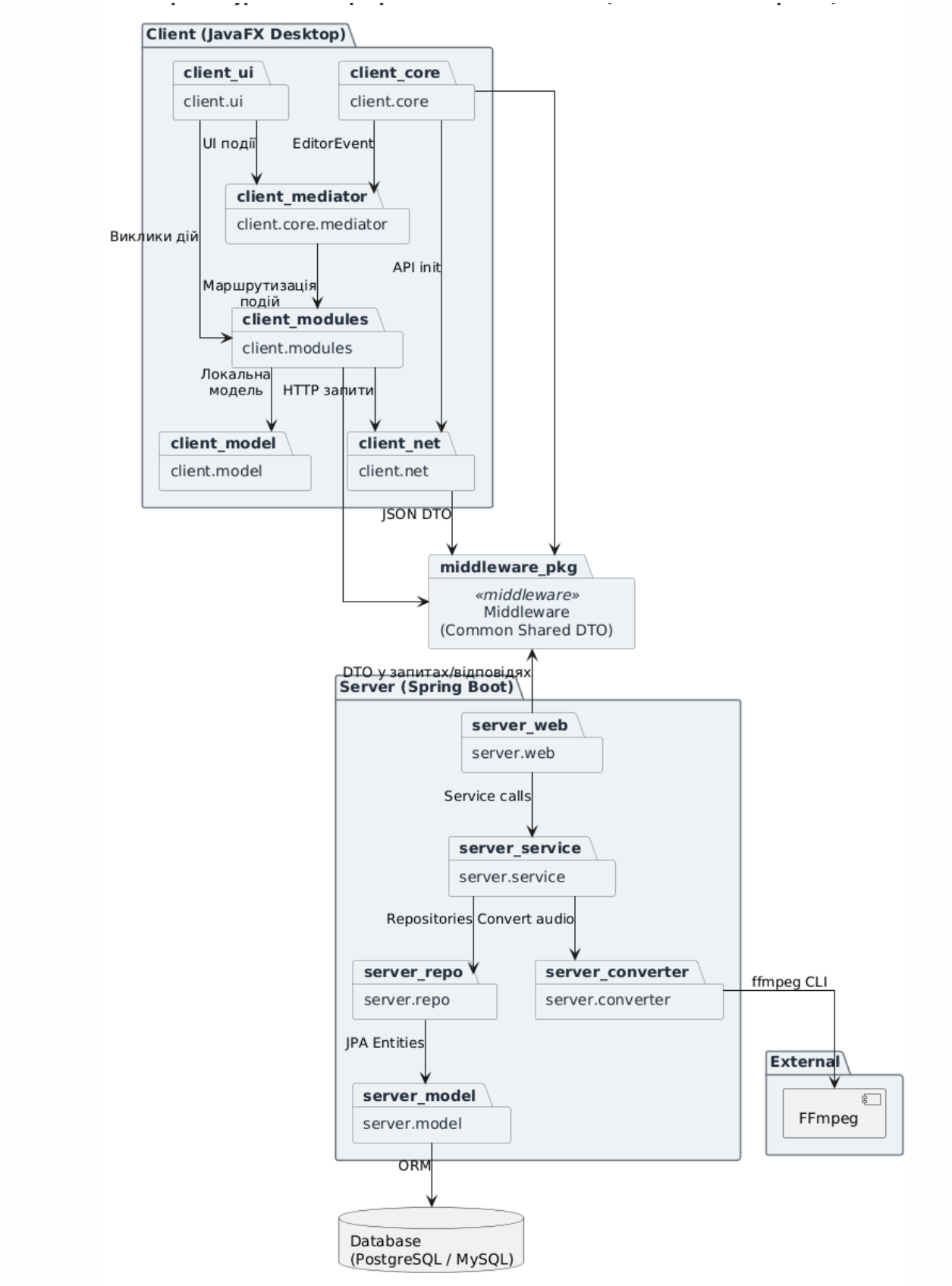

Рисунок 3.1 – Архітектура клієнт–серверної системи Audio Editor на рівні пакетів

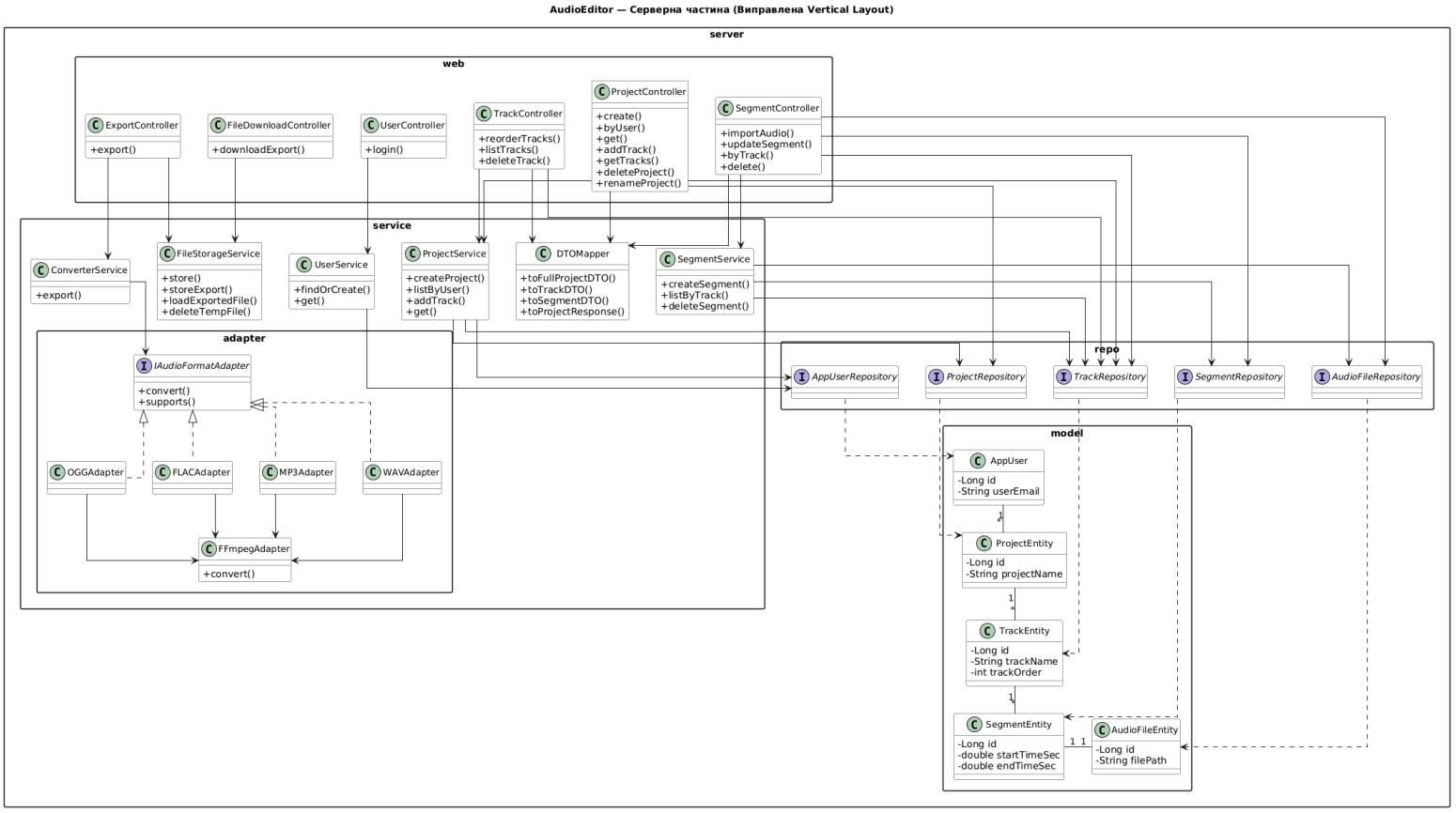

Рисунок 3.2 – Діаграма класів серверної частини системи AudioEditor

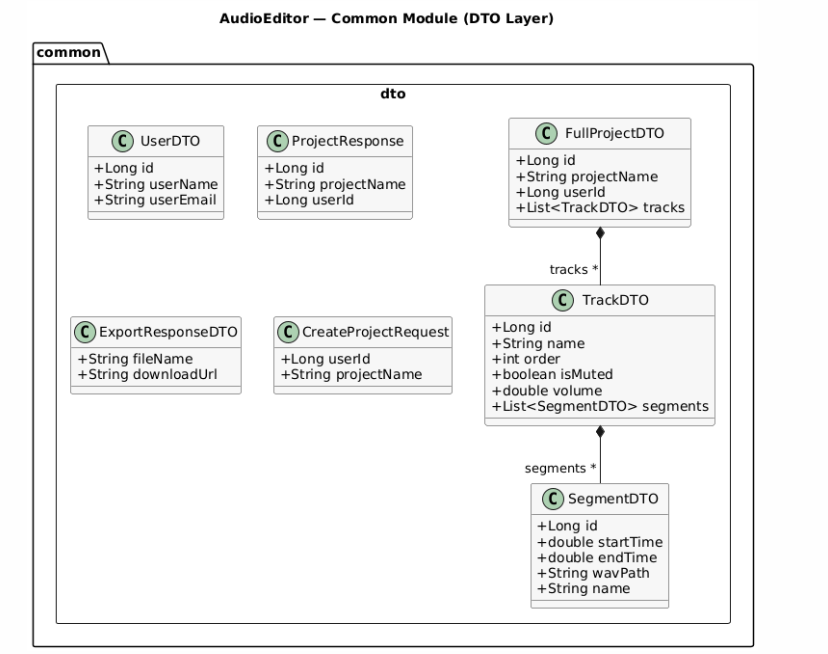

Рисунок 3.3 – Діаграма класів рівня спільного модуля (Common) із DTO-класами системи AudioEditor

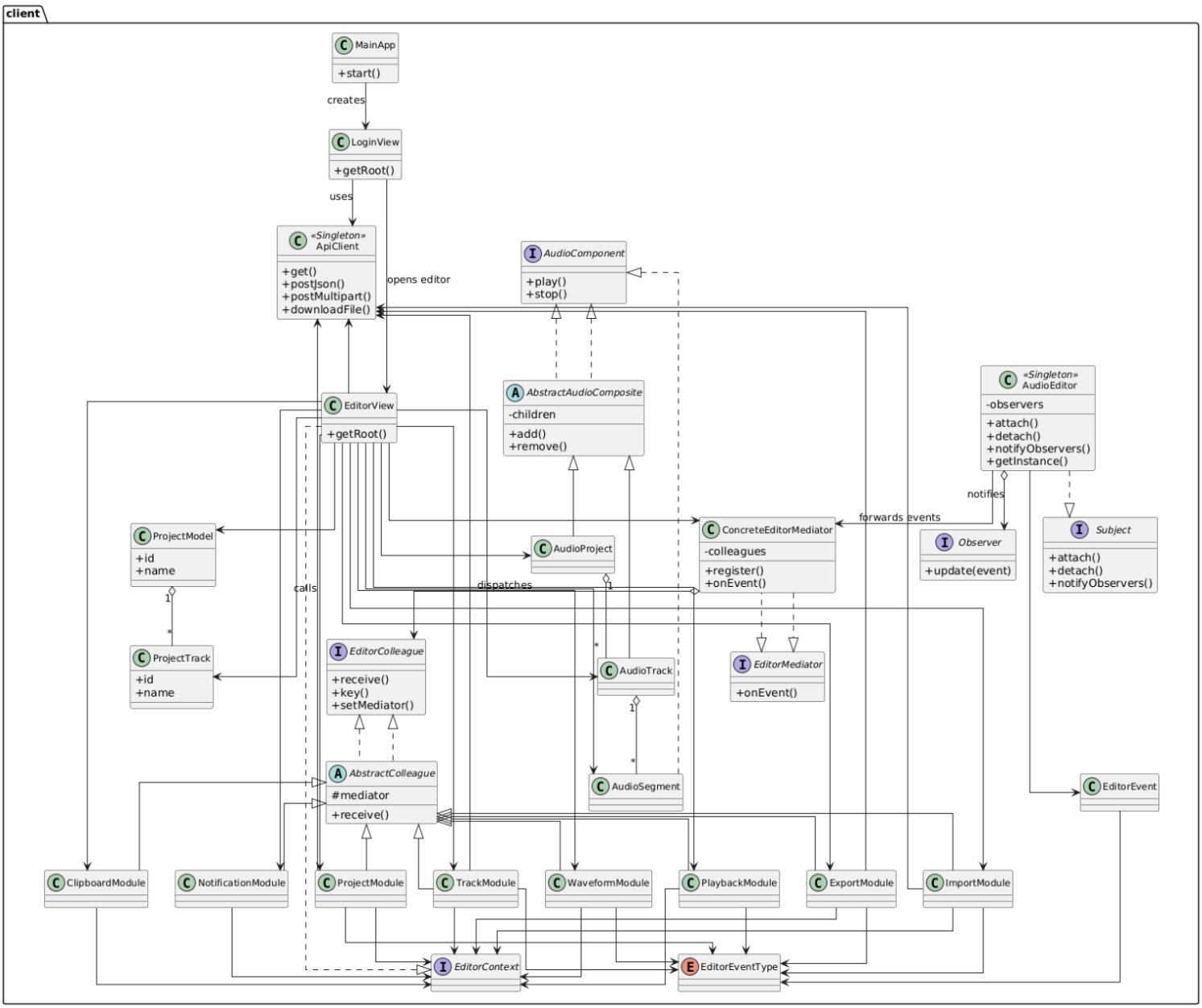

Рисунок 3.4 – Діаграма класів клієнтської частини системи AudioEditor

4. Фрагменти програмного коду, що відображають архітектуру клієнт–серверної системи.

Серверна частина — ProjectController:

```
public class ProjectController {

    private final ProjectService service;
    private final TrackRepository tracks;
```

```
    private final ProjectRepository projectRepo;
    private final DTOMapper mapper;

    public ProjectController(ProjectService service, TrackRepository tracks, ProjectRepository
projectRepo, DTOMapper mapper) {
        this.service = service;
        this.tracks = tracks;
        this.projectRepo = projectRepo;
        this.mapper = mapper;
    }

    @PostMapping
    public ProjectResponse create(@RequestBody CreateProjectRequest req) {
        var project = service.createProject(req.userId(), req.projectName());
        return mapper.toProjectResponse(project);
    }

    @GetMapping("/by-user/{userId}")
    public List<ProjectResponse> byUser(@PathVariable("userId") Long userId) {
        return service.listByUser(userId).stream()
                .map(mapper::toProjectResponse)
                .collect(Collectors.toList());
    }

    @GetMapping("/{projectId}")
    public FullProjectDTO get(@PathVariable("projectId") Long projectId) {
        ProjectEntity project = service.get(projectId);
        return mapper.toFullProjectDTO(project);
    }

    @PostMapping("/{projectId}/tracks")
    public TrackDTO addTrack(@PathVariable("projectId") Long projectId, @RequestParam("name")
String name) {
        TrackEntity track = service.addTrack(projectId, name);
        return mapper.toTrackDTO(track);
    }

    @GetMapping("/{projectId}/tracks")
    public List<TrackDTO> getTracks(@PathVariable("projectId") Long projectId) {
        ProjectEntity p = service.get(projectId);
        return tracks.findByProjectOrderByTrackOrderAsc(p).stream()
                .map(mapper::toTrackDTO)
                .collect(Collectors.toList());
    }

    @DeleteMapping("/{projectId}")
    public void deleteProject(@PathVariable Long projectId) {
        projectRepo.deleteById(projectId);
        System.out.println("Deleted project ID: " + projectId);
    }

    @PutMapping("/{projectId}")
    public void renameProject(@PathVariable Long projectId, @RequestParam("name") String
```

```
newName) {
        ProjectEntity project = projectRepo.findById(projectId)
                .orElseThrow(() -> new RuntimeException("Project not found"));

        if (newName != null && !newName.trim().isEmpty()) {
            project.setProjectName(newName.trim());
            projectRepo.save(project);
            System.out.println("Renamed project " + projectId + " to: " + newName);
        }
    }
}
```

Цей контролер визначає REST-інтерфейс для керування структурами проєкту на сервері. Він приймає HTTP-запити від клієнта, викликає бізнес-логіку сервісного рівня, виконує взаємодію з репозиторіями та повертає DTO-об’єкти у форматі JSON. Саме цей клас задає основний набір сервісних операцій — створення, перейменування, видалення та повернення повної структури проєкту разом із доріжками та сегментами — що лежать в основі всієї серверної частини системи AudioEditor.

Серверна частина — ExportController:

```
@RestController
@RequestMapping("/api/export")
public class ExportController {

    private final ConverterService converter;
    private final FileStorageService storage;

    public ExportController(ConverterService converter, FileStorageService storage) {
        this.converter = converter;
        this.storage = storage;
    }

    @PostMapping
    public ResponseEntity<?> exportFile(
            @RequestParam("userId") Long userId,
            @RequestParam("projectId") Long projectId,
            @RequestParam("format") String format,
            @RequestParam("trackName") String trackName,
            @RequestParam("file") MultipartFile file
    ) {
        File uploaded = null;
        try {
```

```
            uploaded = storage.saveTempFile(file);
            String baseName = trackName.replaceAll("[^a-zA-Z0-9._() -]+", "_");
            File exported = converter.export(uploaded, format, baseName);

            String fileName = exported.getName();
            String urlPath = "/exports/" + fileName;
            return ResponseEntity.ok(new ExportResponseDTO(fileName, urlPath));

        } catch (Exception e) {
            return ResponseEntity.internalServerError().body(Map.of("error", e.getMessage()));
        } finally {
            if (uploaded != null) {
                storage.deleteTempFile(uploaded);
            }
        }
    }
}
```

Контролер відповідає за серверну частину експорту аудіо. Він приймає від клієнта завантажений тимчасовий WAV-файл, викликає сервіс конвертації ffmpeg, формує готовий аудіофайл у потрібному форматі та повертає шлях для завантаження. Це ключовий елемент, що реалізує важливу функцію обробки мультимедійних даних.

Серверна частина — FileDownloadController:

```
@Controller
@RequestMapping("/api")
@CrossOrigin
public class FileDownloadController {

    @Value("${storage.uploadsDir:storage/uploads}")
    private String uploadsDir;

    private final FileStorageService storageService;
    public FileDownloadController(FileStorageService storageService) {
        this.storageService = storageService;
    }

    @GetMapping("/uploads/{filename:.+}")
    @ResponseBody
    public ResponseEntity<Resource> downloadSourceFile(@PathVariable String filename) {
        try {
            File dir = new File(uploadsDir).getAbsoluteFile();
            File file = new File(dir, filename);

            if (!file.exists()) {
                return ResponseEntity.notFound().build();
```

```
        }

        InputStreamResource resource = new InputStreamResource(new FileInputStream(file));
        String contentType = Files.probeContentType(file.toPath());
        if (contentType == null) contentType = "application/octet-stream";

        return ResponseEntity.ok()
                .contentLength(file.length())
                .contentType(MediaType.parseMediaType(contentType))
                .body(resource);

    } catch (Exception e) {
        return ResponseEntity.internalServerError().build();
    }
  }

  @GetMapping("/exports/{filename:.+}")
  @ResponseBody
  public ResponseEntity<Resource> downloadExported(@PathVariable String filename) {
    try {
        File file = storageService.loadExportedFile(filename);
        InputStreamResource resource = new InputStreamResource(new FileInputStream(file));
        String contentType = Files.probeContentType(file.toPath());
        if (contentType == null) contentType = "application/octet-stream";

        return ResponseEntity.ok()
                .contentLength(file.length())
                .contentType(MediaType.parseMediaType(contentType))
                .body(resource);
    } catch (Exception e) {
        return ResponseEntity.notFound().build();
    }
  }
}
```

Цей контролер забезпечує доступ клієнта до збережених файлів. Він повертає оригінальні імпортовані WAV-файли та фінальні експортовані аудіофайли у відповідь на HTTP-запит. Фактично це серверний файловий шлюз, що забезпечує безпечну передачу мультимедіа між додатком і фізичним файловим сховищем.

Серверна частина — SegmentController:

```
@RestController
@RequestMapping("/api/segments")
@CrossOrigin
public class SegmentController {
```

```
private final SegmentRepository segments;
private final TrackRepository tracks;
private final AudioFileRepository audioFiles;
private final DTOMapper mapper;

@Value("${storage.uploadsDir:storage/uploads}")
private String uploadsDir;

public SegmentController(SegmentRepository segments,
                         TrackRepository tracks,
                         AudioFileRepository audioFiles,
                         DTOMapper mapper) {
    this.segments = segments;
    this.tracks = tracks;
    this.audioFiles = audioFiles;
    this.mapper = mapper;
}

@PostMapping("/import/{trackId}")
public SegmentDTO importAudio(@PathVariable Long trackId,
                              @RequestParam("file") MultipartFile file) throws IOException {
    TrackEntity track = tracks.findById(trackId)
            .orElseThrow(() -> new RuntimeException("Track not found"));

    File savedFile = saveFileToStorage(file);

    AudioFileEntity audioEntity = new AudioFileEntity();
    audioEntity.setFileName(savedFile.getName());
    audioEntity.setFilePath(savedFile.getAbsolutePath());
    audioEntity.setFileFormat("wav");
    audioFiles.save(audioEntity);

    SegmentEntity segment = new SegmentEntity();
    segment.setTrack(track);
    segment.setAudioFile(audioEntity);
    segment.setStartTimeSec(0.0);
    segment.setEndTimeSec(0.0);

    SegmentEntity savedSegment = segments.save(segment);
    return mapper.toSegmentDTO(savedSegment);
}

@PostMapping("/{id}/upload")
public ResponseEntity<?> updateSegmentAudio(@PathVariable Long id,
                                            @RequestParam("file") MultipartFile file) {
    try {
        System.out.println("--- SAVE REQUEST for Segment " + id + " ---");
        SegmentEntity segment = segments.findById(id)
                .orElseThrow(() -> new RuntimeException("Segment not found"));

        File savedFile = saveFileToStorage(file);

        AudioFileEntity audio = segment.getAudioFile();
```

```
            if (audio == null) {
                audio = new AudioFileEntity();
                segment.setAudioFile(audio);
            }
            audio.setFileName(savedFile.getName());
            audio.setFilePath(savedFile.getAbsolutePath());

            audioFiles.save(audio);
            segments.save(segment);

            System.out.println("Database updated with new file: " + savedFile.getName());
            return ResponseEntity.ok().body("Saved");
        } catch (Exception e) {
            e.printStackTrace();
            return ResponseEntity.internalServerError().body(e.getMessage());
        }
    }

    @GetMapping("/by-track/{trackId}")
    public List<SegmentDTO> byTrack(@PathVariable Long trackId) {
        TrackEntity t = tracks.findById(trackId).orElseThrow();
        return segments.findByTrackOrderByStartTimeSecAsc(t).stream()
                .map(mapper::toSegmentDTO)
                .collect(Collectors.toList());
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        segments.deleteById(id);
    }

    private File saveFileToStorage(MultipartFile file) throws IOException {
        File dir = new File(uploadsDir).getAbsoluteFile();
        if (!dir.exists()) {
            boolean created = dir.mkdirs();
            System.out.println("Created uploads directory: " + dir.getAbsolutePath() + " -> " + created);
        }

        String original = file.getOriginalFilename();
        String ext = ".wav";
        if (original != null && original.contains(".")) {
            ext = original.substring(original.lastIndexOf("."));
        }

        String uniqueName = UUID.randomUUID().toString() + ext;

        File dest = new File(dir, uniqueName);
        file.transferTo(dest);

        System.out.println("File saved physically to: " + dest.getAbsolutePath());
        return dest;
    }
}
```

SegmentController реалізує логіку роботи з окремими аудіосегментами: імпорт файлів, збереження оновлених версій, отримання списку сегментів доріжки та видалення. Саме він приймає завантажені файли, розміщує їх у сховищі та оновлює записи в базі даних. Таким чином він забезпечує синхронізацію звукового вмісту між клієнтом і сервером.

Серверна частина — TrackController:

```
@RestController
@RequestMapping("/api/tracks")
@CrossOrigin
public class TrackController {

    private final TrackRepository repo;
    private final ProjectService projects;
    private final DTOMapper mapper;

    public TrackController(TrackRepository repo, ProjectService projects, DTOMapper mapper) {
        this.repo = repo;
        this.projects = projects;
        this.mapper = mapper;
    }

    @GetMapping("/by-project/{projectId}")
    public ResponseEntity<?> byProject(@PathVariable Long projectId) {
        try {
            ProjectEntity p = projects.get(projectId);
            List<TrackEntity> list = repo.findByProjectOrderByTrackOrderAsc(p);
            List<TrackDTO> dtos = list.stream()
                    .map(mapper::toTrackDTO)
                    .collect(Collectors.toList());

            return ResponseEntity.ok(dtos);
        } catch (Exception e) {
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(e.getMessage());
        }
    }

    @DeleteMapping("/{trackId}")
    public void deleteTrack(@PathVariable Long trackId) {
        repo.deleteById(trackId);
        System.out.println("Deleted track ID: " + trackId);
    }
}
```

Контролер керує треками проєкту: повертає їх список у правильному порядку та дозволяє їх видаляти. Він є частиною загальної ієрархії «Проєкт -- Трек --

Сегмент» і забезпечує коректну структуризацію даних та синхронізацію стану треків між сервером і клієнтом.

Серверна частина — UserController:

```
@RestController
@RequestMapping("/api/users")
@CrossOrigin
public class UserController {

    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @PostMapping("/find-or-create")
    public ResponseEntity<?> findOrCreate(
            @RequestParam("name") String name,
            @RequestParam("email") String email) {

        if (name == null || name.trim().isEmpty() || email == null || email.trim().isEmpty()) {
            return ResponseEntity.badRequest().body("Name and email cannot be empty");
        }

        if (!email.matches("^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$")) {
            return ResponseEntity.badRequest().body("Invalid email format");
        }

        AppUser user = userService.findOrCreate(name.trim(), email.trim());
        return ResponseEntity.ok(new UserDTO(user.getId(), user.getUserName(),
user.getUserEmail()));
    }
}
```

UserController відповідає за створення й пошук користувачів. Клієнт передає ім’я та email, а сервер повертає унікальний ідентифікатор користувача. Це фундаментальна точка входу для персоналізації, збереження проєктів і подальшої роботи в системі.

Клієнтська частина — ApiClient:

```
public class ApiClient {
```

```
    private static ApiClient INSTANCE;

    private final String base;
    private final HttpClient http = HttpClient.newBuilder()
            .version(HttpClient.Version.HTTP_1_1)
            .build();
    private final ObjectMapper mapper = new ObjectMapper();

    private ApiClient(String baseUrl) {
        this.base = baseUrl.endsWith("/") ? baseUrl.substring(0, baseUrl.length() - 1) : baseUrl;
    }

    public static synchronized ApiClient getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new ApiClient("http://localhost:8080/api");
        }
        return INSTANCE;
    }
    public String getBaseUrl() {
        return base;
    }

    private HttpResponse<String> safeSend(HttpRequest req)
            throws IOException, InterruptedException {
        try {
            return http.send(req, HttpResponse.BodyHandlers.ofString());
        } catch (java.net.ConnectException e) {
            throw new RuntimeException("Сервер недоступний за адресою: " + base +
                    "\nПереконайся, що Spring Boot запущено (порт 8080).", e);
        }
    }

    public String postJson(String path, Object body) throws Exception {
        String json = mapper.writeValueAsString(body);
        HttpRequest req = HttpRequest.newBuilder(URI.create(base + path))
                .header("Content-Type", "application/json")
                .POST(HttpRequest.BodyPublishers.ofString(json))
                .build();
        HttpResponse<String> res = safeSend(req);
        if (res.statusCode() >= 400)
            throw new RuntimeException("POST " + path + " -> " + res.statusCode() + ": " +
res.body());
        return res.body();
    }

    public String postForm(String path, String query) throws Exception {
        String encodedQuery = query.contains("&") ? query : encodeQuery(query);
        HttpRequest req = HttpRequest.newBuilder(URI.create(base + path))
                .header("Content-Type", "application/x-www-form-urlencoded")
                .POST(HttpRequest.BodyPublishers.ofString(encodedQuery))
                .build();
        HttpResponse<String> res = safeSend(req);
        if (res.statusCode() >= 400)
            throw new RuntimeException("POST " + path + " -> " + res.statusCode() + ": " +
res.body());
        return res.body();
    }

    public String get(String path) throws Exception {
        HttpRequest req = HttpRequest.newBuilder(URI.create(base + path))
                .GET()
                .build();
        HttpResponse<String> res = safeSend(req);
        if (res.statusCode() >= 400)
            throw new RuntimeException("GET " + path + " -> " + res.statusCode() + ": " + res.body());
        return res.body();
```

```
    }

    public String postMultipart(String path, Map<String, String> params, File file) throws Exception {
        String boundary = "===" + System.currentTimeMillis() + "===";
        var byteStream = new ByteArrayOutputStream();
        var writer = new PrintWriter(new OutputStreamWriter(byteStream, StandardCharsets.UTF_8),
true);
        String LINE_FEED = "\r\n";

        for (var entry : params.entrySet()) {
            writer.append("--").append(boundary).append(LINE_FEED);
            writer.append("Content-Disposition: form-data;
name=\"").append(entry.getKey()).append("\"").append(LINE_FEED);
            writer.append("Content-Type: text/plain; charset=UTF-8").append(LINE_FEED);
            writer.append(LINE_FEED).append(entry.getValue()).append(LINE_FEED);
            writer.flush();
        }

        writer.append("--").append(boundary).append(LINE_FEED);
        writer.append("Content-Disposition: form-data; name=\"file\"; filename=\"")
                .append(file.getName()).append("\"").append(LINE_FEED);
        writer.append("Content-Type: ").append(Files.probeContentType(file.toPath()))
                .append(LINE_FEED);
        writer.append(LINE_FEED);
        writer.flush();

        Files.copy(file.toPath(), byteStream);
        byteStream.flush();
        writer.append(LINE_FEED).flush();
        writer.append("--").append(boundary).append("--").append(LINE_FEED);
        writer.close();

        HttpRequest request = HttpRequest.newBuilder(URI.create(base + path))
                .header("Content-Type", "multipart/form-data; boundary=" + boundary)
                .POST(HttpRequest.BodyPublishers.ofByteArray(byteStream.toByteArray()))
                .build();

        HttpResponse<String> response = safeSend(request);
        if (response.statusCode() >= 400) {
            throw new RuntimeException("POST multipart " + path + " -> " +
                    response.statusCode() + ": " + response.body());
        }
        return response.body();
    }

    private String encodeQuery(String query) {
        if (query == null || !query.contains("=")) return query;
        String[] parts = query.split("=", 2);
        String key = parts[0];
        String value = parts.length > 1 ? parts[1] : "";
        return key + "=" + URLEncoder.encode(value, StandardCharsets.UTF_8);
    }

    public void downloadFile(String path, File destination) throws Exception {
        String[] parts = path.split("/");
        StringBuilder encodedPath = new StringBuilder();

        for (int i = 0; i < parts.length; i++) {
            if (!parts[i].isEmpty()) {

                String encodedPart = URLEncoder.encode(parts[i], StandardCharsets.UTF_8)
                        .replace("+", "%20");
                encodedPath.append("/").append(encodedPart);
            }
        }
```

```
        String finalPath = encodedPath.toString();
        if (!path.startsWith("/") && finalPath.startsWith("/")) {
            finalPath = finalPath.substring(1);
        }

        URI uri = URI.create(base + finalPath);

        java.net.http.HttpRequest req = java.net.http.HttpRequest.newBuilder(uri)
                .GET()
                .build();

        java.net.http.HttpResponse<java.io.InputStream> res = http.send(req,
java.net.http.HttpResponse.BodyHandlers.ofInputStream());

        if (res.statusCode() >= 400) {
            throw new RuntimeException("Download failed: " + res.statusCode());
        }

        try (java.io.InputStream in = res.body();
             java.io.FileOutputStream out = new java.io.FileOutputStream(destination)) {
            in.transferTo(out);
        }
    }
    public void delete(String path) throws Exception {
        java.net.http.HttpRequest req = java.net.http.HttpRequest.newBuilder(java.net.URI.create(base +
path))
                .DELETE()
                .build();

        java.net.http.HttpResponse<String> res = http.send(req,
java.net.http.HttpResponse.BodyHandlers.ofString());

        if (res.statusCode() >= 400) {
            throw new RuntimeException("DELETE " + path + " -> " + res.statusCode() + ": " +
res.body());
        }
    }

    public void put(String path) throws Exception {
        java.net.http.HttpRequest req = java.net.http.HttpRequest.newBuilder(java.net.URI.create(base +
path))
                .PUT(java.net.http.HttpRequest.BodyPublishers.noBody())
                .build();

        java.net.http.HttpResponse<String> res = http.send(req,
java.net.http.HttpResponse.BodyHandlers.ofString());

        if (res.statusCode() >= 400) {
            throw new RuntimeException("PUT " + path + " -> " + res.statusCode() + ": " + res.body());
        }
    }
}
```

ApiClient — це універсальний HTTP-клієнт для десктопної JavaFX-програми. Він інкапсулює логіку формування запитів (GET, POST, PUT, DELETE), серіалізацію JSON-даних, завантаження та відправлення файлів, обробку помилок та адресацію до REST-інтерфейсу сервера. Цей клас – центральна точка зв’язку клієнтської частини з backend-сервісами.

Виконує POST-запит із JSON-тілом. Метод використовується для створення проєктів, відправлення DTO, оновлення сегментів та інших дій, де потрібна структурована інформація.

Метод для отримання даних від сервера: проєктів, треків, сегментів та метаданих. Формує основу клієнтського читання структури системи.

postMultipart – відправляє файли (наприклад аудіосегменти) у форматі multipart/form-data. Це ключовий механізм для імпорту та збереження звукових даних на сервері.

DELETE / PUT – забезпечують можливість видалення або оновлення сутностей (проєктів, треків, сегментів). Завдяки цим методам клієнт повноцінно керує станом серверних даних.

5. Візуальна частина реалізованого клієнтського застосунку

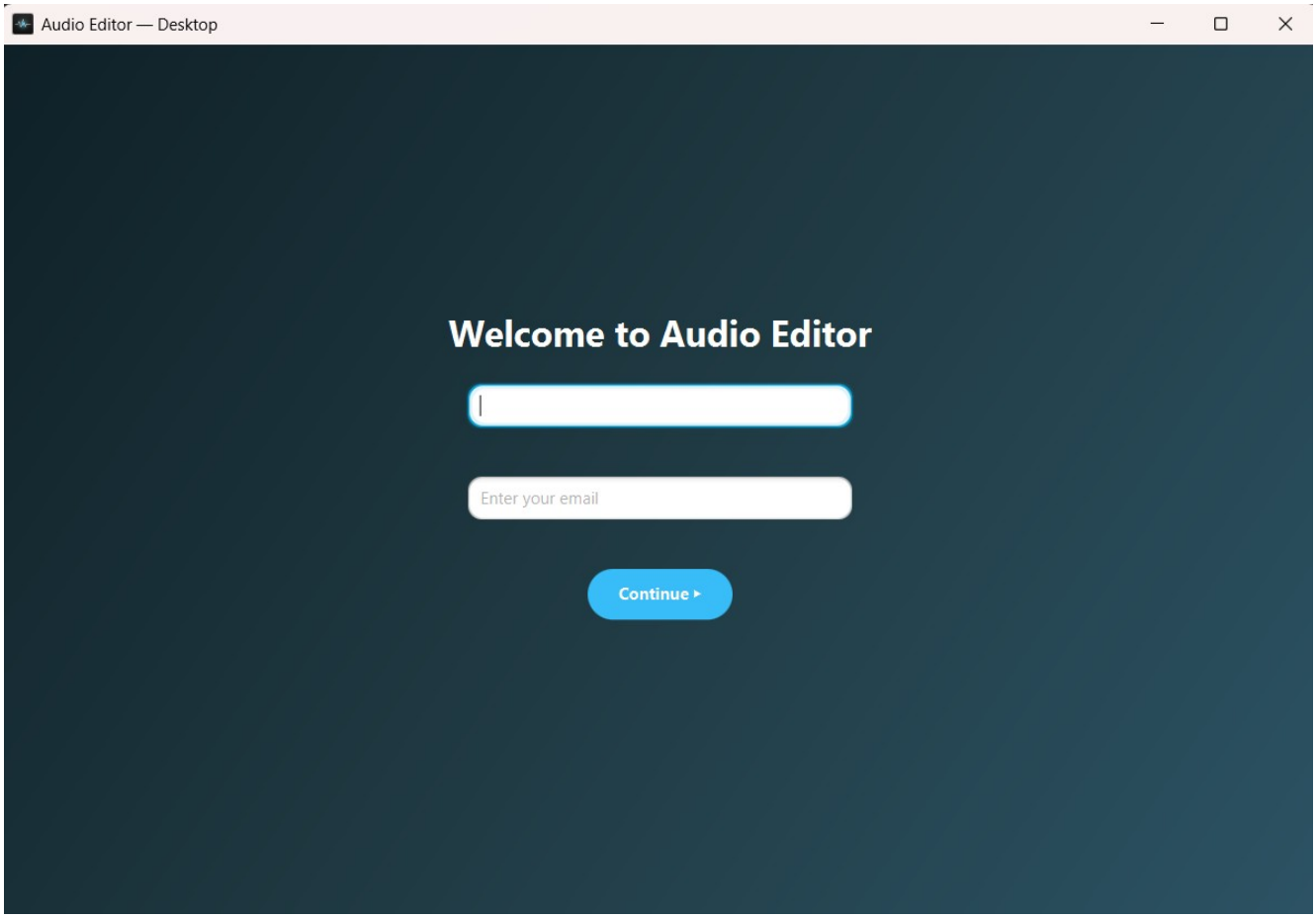

Рисунок 5.1 – Вікно авторизації користувача (LoginView)

Рисунок 5.2 – Інтерфейс клієнтського застосунку AudioEditor

Відповіді на контрольні запитання:

1. Що таке клієнт-серверна архітектура?

Клієнт-серверна архітектура – це найпростіший варіант розподілених додатків, де виділяються два основні види додатків: клієнти (представляють додаток користувачеві) і сервери (використовуються для зберігання та обробки даних).

Розрізняють два основні типи клієнтів:

- *Тонкий клієнт: Передає більшість логіки роботи на обробку на сервер, а сам зберігає лише візуальне уявлення одержуваних відповідей (наприклад, класичні Web-застосунки).*

- *Товстий клієнт: Більшість логіки обробки даних містить на своїй стороні, що розвантажує сервер (наприклад, мобільні або десктопні застосунки).*

Клієнт-серверна взаємодія часто організовується за допомогою 3-рівневої структури: клієнтська частина, загальна частина (middleware) та серверна частина.

2. Розкажіть про сервіс-орієнтовану архітектуру.

Сервіс-орієнтована архітектура (SOA) — це модульний підхід до розробки програмного забезпечення, заснований на використанні розподілених, слабо пов'язаних (Loose coupling) сервісів або служб. Ці служби оснащені стандартизованими інтерфейсами для взаємодії за стандартизованими протоколами.

Комплекси, розроблені відповідно до SOA, зазвичай реалізуються як набір веб-служб, які надають певні бізнес-функції (наприклад, отримання інформації про наявність матеріалів на складі).

3. Якими принципами керується SOA?

SOA керується принципами, що випливають з її модульного та розподіленого характеру:

- *Слабка зв'язаність (Loose coupling): Сервіси є слабо пов'язаними між собою.*
- *Архітектура заснована на використанні розподілених сервісів.*
- *Сервіси оснащені стандартизованими інтерфейсами для взаємодії.*
- *Взаємодія здійснюється за стандартизованими протоколами , такими як HTTP з використанням SOAP або REST.*

- *Сервіси спілкуються між собою виключно за рахунок обміну повідомленнями, а не через спеціальні інтеграції (наприклад, прямий доступ до однієї бази даних).*

4. Як між собою взаємодіють сервіси в SOA?

Сервіси в SOA взаємодіють між собою шляхом обміну повідомленнями. Взаємодія, як правило, відбувається по протоколу HTTP з використанням SOAP або REST. Часто для обміну даними використовується централізований програмний компонент — шину даних (Enterprise Service Bus).

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

Пошук сервісів: Згідно з SOA, сервіси реєструються на спеціальних сервісах. Будь-яка команда розробників, якій потрібен доступ, може знайти ці зареєстровані сервіси та використовувати їх.

Запити: Запити до сервісів робляться з використанням стандартизованих протоколів (наприклад, HTTP) та форматів (SOAP або REST) через їхні стандартизовані інтерфейси.

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

Переваги

- *Забезпечується централізоване управління обліковими записами користувачів, безпекою та доступом, що спрощує мережне адміністрування. Користувачеві потрібен один пароль для входу в мережу і для отримання доступу до всіх ресурсів.*
- *Централізоване зберігання даних.*

- *Простота розгортання та оновлення (для тонких клієнтів): Оновлювати потрібно лише сервери, і клієнти з наступними запитами автоматично працюватимуть з оновленою системою.*
- *Клієнти можуть бути «тонкими»: Клієнт зберігає лише візуальне уявлення одержуваних від сервера відповідей, передаючи більшість операцій на обробку серверу.*
- *Ефективний доступ до мережевих ресурсів.*
- *Дозволяє організовувати мережі з великою кількістю робочих станцій.*
- *Розвантаження сервера (для товстих клієнтів): Більшість логіки обробки даних міститься на стороні клієнта, що сильно розвантажує сервер.*

Недоліки

- *Несправність сервера може зробити мережу непрацездатною, як мінімум призвести до втрати мережевих ресурсів.*
- *Потреба у постійному мережевому з'єднанні (для тонких клієнтів): Такі застосунки не працюють, якщо сервер не доступний.*
- *Можлива перевантаженість сервера (для тонких клієнтів): Майже все навантаження лягає на сервер або групу серверів.*
- *Вища вартість мереж і мережевого обладнання.*
- *Вимагають кваліфікованого персоналу для адміністрування.*
- *Складність оновлення (для товстих клієнтів): Оновлення товстих клієнтів (наприклад, мобільних застосунків) складніше, ніж для тонких, оскільки потребує інсталяції.*

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

Переваги P2P Моделі:

- *Децентралізація: Відсутність центрального сервера, що зменшує залежність від одного вузла.*
- *Підвищується стійкість мережі до збоїв і атак.*
- *Кожен вузол має рівні права та може виконувати одночасно функції клієнта (отримувати ресурси) і сервера (надавати ресурси).*
- *Вузли надають доступ до своїх власних ресурсів, таких як обчислювальна потужність, дисковий простір або файли.*

Недоліки P2P Моделі:

- *Через децентралізацію складно контролювати дані, які передаються.*
- *Синхронізація даних.*
- *Ефективність пошуку даних знижується зі збільшенням кількості вузлів у мережі. Для підвищення ефективності пошуку потрібно застосовувати спеціальні алгоритми.*
- *Складно контролювати дані, які передаються через відсутність централізації.*

8. Що таке мікро-сервісна архітектура?

Мікро-сервісна архітектура — це підхід до створення серверного додатку як набору невеликих, незалежних служб. Вона розглядається як подальший розвиток сервіс-орієнтованої архітектури (SOA).

Кожен мікросервіс працює у власному процесі, реалізує конкретну бізнес-функцію в межах обмеженого контексту та розробляється і розгортається автономно від інших сервісів. Взаємодія між сервісами здійснюється через легковагові протоколи — HTTP/HTTPS, REST, WebSockets, а також через системи обміну повідомленнями, зокрема AMQP.

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

У мікросервісній архітектурі служби взаємодіють між собою через стандартизовані та незалежні мережеві протоколи. Найпоширенішими є:

- *HTTP / HTTPS*
- *WebSockets*
- *AMQP*

Додатково в мікросервісах часто використовують також:

- *gRPC (HTTP/2) — високопродуктивний протокол з бінарною передачею даних*
- *MQTT — для IoT-орієнтованих сервісів*
- *Kafka protocol — для масштабованих систем потокової обробки подій*

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 не є сервіс-орієнтованою архітектурою (SOA). У цьому випадку шар бізнес-логіки, який реалізується у вигляді "сервісних" класів всередині застосунку, належить до звичайної багаторівневої (layered) архітектури, а не до SOA.

У сервіс-орієнтованій архітектурі сервіси повинні бути розподіленими, незалежними компонентами, що:

- *працюють у різних процесах або навіть на різних вузлах;*
- *мають слабу зв'язаність та стандартизовані інтерфейси;*
- *взаємодіють через протоколи типу HTTP/S, SOAP або REST;*
- *можуть розгортатися й оновлюватися незалежно один від одного.*

Внутрішній шар сервісів у монолітному застосунку не має цих властивостей, оскільки це просто частина одного виконуваного модуля, який розгортається та працює як єдине ціле. Отже, цей підхід не відповідає ключовим принципам SOA.

Висновки:

У ході виконання лабораторної роботи було створено повноцінну клієнт–серверну систему AudioEditor, яка реалізує сучасний підхід до побудови розподілених архітектур. Сформована система складається з трьох структурних частин — клієнта (JavaFX), спільного модуля (DTO), та сервера (Spring Boot), — що взаємодіють між собою через REST API. Така організація забезпечила чітке розділення відповідальностей, низьку зв'язність компонентів і гнучкість у подальшому масштабуванні.

У клієнтській частині застосовано декілька патернів проєктування (Composite, Mediator, Observer), що дозволило побудувати модульну, керовану подіями архітектуру з логічно розділеними підсистемами редагування аудіо, обробки подій, візуалізації та відтворення. Реалізована ієрархічна модель аудіопроєкту забезпечила зручність роботи з треками й сегментами, а використання патерну Mediator зменшило зв'язність між функціональними модулями клієнта. Завдяки ApiClient клієнтська частина здійснює повний спектр HTTP-запитів, включно з JSON, multipart/form-data, GET/POST/PUT/DELETE, що відтворює реальну комунікацію у розподілених системах.

У спільному модулі сформовано набір DTO-класів, які виступають єдиним транспортним форматом між клієнтом і сервером. Використання DTO дозволило ізолювати внутрішні серверні сутності від зовнішнього API, зберегти стабільність контрактів та забезпечити відповідність принципам чистої архітектури.

Серверну частину побудовано на основі Spring Boot. Використано багатошарову модель, у якій контролери відповідають за приймання запитів, сервіси містять бізнес-логіку, а репозиторії забезпечують взаємодію з базою даних. Реалізовано операції створення, оновлення, видалення і завантаження аудіопроєктів, треків та сегментів, а також механізм імпорту й експорту аудіофайлів. Уся інформація зберігається в базі даних, що дозволяє повністю відновити структуру проєкту під час повторного входу користувача в систему.

Узгоджена взаємодія між усіма компонентами системи, використання патернів проєктування, стандартизованих API та архітектурних практик дозволили створити масштабований, структуровано організований і функціонально багатий застосунок. Результати роботи підтверджують можливість інтеграції складного редактора аудіо з серверною платформою, що виконує функції збереження даних та управління ними у розподіленому середовищі.

Таким чином, поставлені завдання повністю виконано: спроєктовано архітектуру системи, реалізовано клієнтську, серверну та middleware-частини, створено механізм взаємодії між компонентами, а також наведено діаграму класів та фрагменти коду, що відображають ключові архітектурні рішення.