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

Лабораторна робота № 9

із дисципліни «Технології розроблення програмного забезпечення»

Тема: «Взаємодія компонентів системи»

Виконав
Студент групи ІА-31:
Козир Я. О.

Перевірив:
Мягкий М. Ю.

Київ 2025

Зміст

1. **Мета:**

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

2. **Теоретичні відомості**

Клієнт-серверна архітектура — модель, де клієнт відповідає за взаємодію з користувачем, а сервер — за зберігання та обробку даних. Тонкий клієнт (наприклад, вебзастосунки) передає більшість операцій на сервер, спрощуючи оновлення. Товстий клієнт (мобільні або десктопні програми) виконує логіку локально, зменшуючи навантаження на сервер і дозволяючи працювати офлайн. SPA (Single Page Application) — проміжний варіант: логіка на клієнті, але робота можлива лише з підключенням до сервера. Типова структура включає три рівні: клієнтський (інтерфейс), спільний (middleware) і серверний (бізнес-логіка та дані).

Peer-to-Peer (P2P) — децентралізована модель, де кожен вузол одночасно є клієнтом і сервером. Усі учасники рівноправні, обмінюються ресурсами без центрального сервера (наприклад, BitTorrent, блокчейн, Skype). Недоліки: складність забезпечення безпеки, синхронізації та пошуку даних у великих мережах.

Сервіс-орієнтована архітектура (SOA) — модульний підхід, де система складається з незалежних сервісів зі стандартизованими інтерфейсами (HTTP, SOAP, REST). Сервіси виконують конкретні бізнес-функції, обмінюються повідомленнями і можуть бути інтегровані через Enterprise Service Bus (ESB). SOA стала основою для мікросервісів.

Мікросервісна архітектура — створення додатків як набору незалежних малих сервісів, що взаємодіють через HTTP, WebSockets або AMQP. Кожен мікросервіс має власну логіку, життєвий цикл і може розгортатися автономно. Переваги: гнучкість, масштабованість і легке супроводження великих систем.

3. **Хід роботи**

Тема :

21. **Online radio station** (iterator, adapter, factory method, facade, visitor, client-server)

Додаток повинен служити сервером для радіостанції з можливістю мовлення на радіостанцію (64, 92, 128, 196, 224 kb/s) в потоковому режимі; вести облік підключених користувачів і статистику відвідувань і прослуховувань; налаштувати папки з піснями і можливість вести списки програвання або playlists (не відтворювати всі пісні).

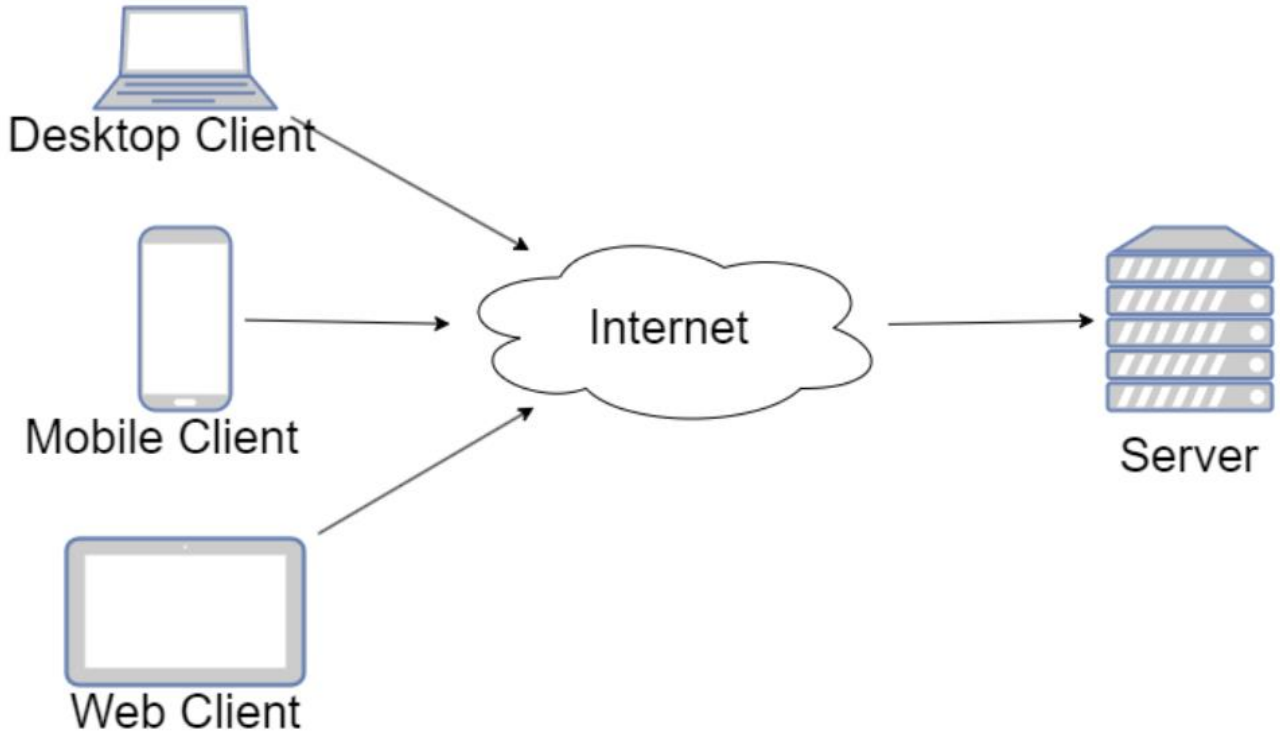

Рис.1 – Архітектура «клієнт-сервер»

Архітектура системи "Online Radio Station"

У даному проєкті реалізовано класичну клієнт-серверну архітектуру, де клієнтом виступає вебінтерфейс користувача (слухача, Діджея чи адміна), а сервер забезпечує всю логіку обробки запитів, конвертацію медіа, зберігання та доступ до даних.

Серверна частина

Серверна частина побудована з використанням ASP.NET Core MVC, що виконує роль основного сервера застосунку. Вона реалізує:

- Обробку HTTP-запитів клієнта через контролери (наприклад, AdminController, DjController, HomeController).

- Складну бізнес-логіку, таку як обробка та конвертація аудіофайлів за допомогою "Фасаду" (AudioProcessingFacade).
- Доступ до бази даних MS SQL Server через репозиторії (StationRepository, TrackRepository, PlaybackQueueRepository), які використовують ORM Entity Framework для взаємодії з базою. База даних зберігає всю інформацію про користувачів, станції, треки та плейлисти.

Клієнтська частина

Клієнтська частина представлена у вигляді вебінтерфейсу, реалізованого за допомогою Razor-сторінок (.cshtml), JavaScript та бібліотеки hls.js для відтворення HLS-стрімів. Вона виконує дві основні функції:

1. Відображення даних: Показує плейлисти, сторінки статистики, дашборди.
2. Надсилання запитів: Передає дані на сервер через HTML-форми (наприклад, форма завантаження MP3-файлу в DjController).

Клієнт не має власної бізнес-логіки — він виконує роль "тонкого клієнта", тобто лише взаємодіє із сервером та відображає результати користувачеві.

Middleware (Посередницький шар)

Посередницький шар (middleware) у проєкті реалізовано у вигляді окремих проектів RadioStation.Domain та RadioStation.Services, які інкапсулюють всю логіку:

- Сервіси (StationService, UserService) та Фасади (AudioProcessingFacade) містять усю бізнес-логіку.
- Інтерфейси патернів (IAudioProcessor, IAudioConverter, IStreamFactory, IRepository) визначають контракти для компонентів системи.
- Моделі (RadioStationEntity, Track, User, DjStream) описують сутності системи, які спільно використовуються всіма шарами.

Цей шар виступає сполучною ланкою між контролерами (які обробляють запити клієнта) та інфраструктурою (базою даних, ffmpeg.exe).

Взаємодія клієнта і сервера

Оскільки система є багатофункціональною, типові послідовності взаємодії виглядають так:

1. Сценарій: Завантаження треку Діджеєм

1. Користувач (Діджей) заповнює форму в Views/Dj/Upload.cshtml (клієнт) і надсилає HTTP POST-запит.
2. DjController (сервер) приймає запит, файл та дані (бітрейт).

3. Контролер викликає "Фасад" IAudioProcessor.ProcessNewTrackAsync.
4. Фасад викликає "Фабрику" (StreamFactory) для створення "продукту" (напр., LowBitrateStream).
5. "Продукт" викликає "Адаптер" (IAudioConverter), який запускає ffmpeg.exe для конвертації.
6. Фасад зберігає результат (HlsUrl) у TrackRepository та PlaybackQueueRepository.
7. Сервер повертає клієнту Redirect на сторінку прослуховування.

2. Сценарій: Прослуховування станції Слухачем

1. Користувач (клієнт) натискає "Слухати" — браузер надсилає HTTP GET-запит.
2. HomeController (сервер) приймає запит і звертається до StationService.
3. StationService викликає StationRepository, який завантажує станцію разом з плейлистом (через .Include).
4. Результат (RadioStationEntity) передається у View Listen.cshtml.
5. View (клієнт) використовує патерн "Ітератор" (model.CreateIterator()) для відображення списку треків, а hls.js (клієнт) відтворює HLS-стрім.

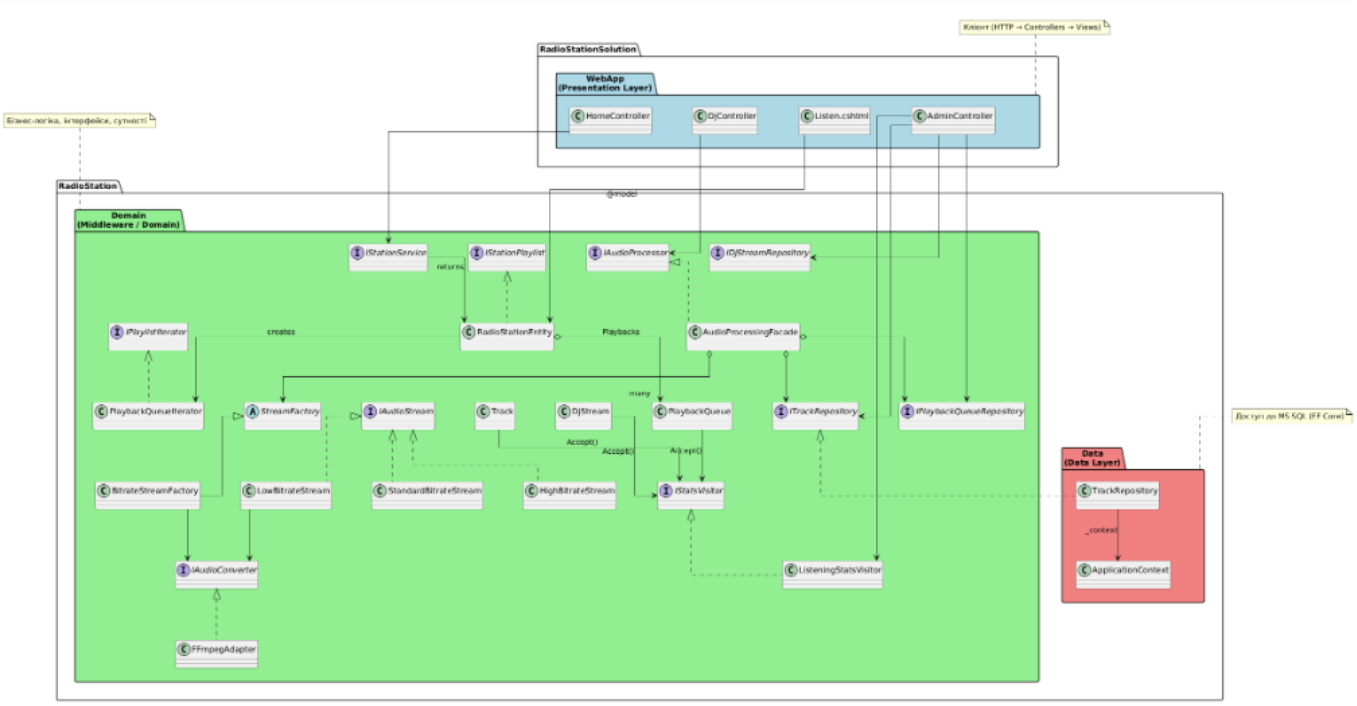

Рис. 2 - Діаграма класів

Архітектура чітко розділена на три логічні шари (пакети):

1. RadioStationSolution.WebApp (Presentation Layer / Клієнт):

- Цей шар відповідає за взаємодію з користувачем. Він виступає в ролі "тонкого клієнта".
- Містить Контролери (DjController, AdminController, HomeController), які приймають HTTP-запити від користувача.
- Містить Представлення (Listen.cshtml), які відображають дані.
- Цей шар не містить бізнес-логіки, а лише делегує всі операції середньому шару (Middleware).

2. RadioStation.Domain (Middleware / Domain):
   - Це "мозок" і ядро всієї системи. Він містить всю бізнес-логіку та контракти (інтерфейси), реалізуючи принцип інверсії залежностей.
   - Тут реалізовано патерни:
     - Facade: AudioProcessingFacade приховує складний процес обробки треку.
     - Factory Method: BitrateStreamFactory створює "продукти" (LowBitrateStream тощо).
     - Adapter: FFmpegAdapter "адаптує" зовнішній ffmpeg.exe до інтерфейсу IAudioConverter.
     - Iterator: RadioStationEntity реалізує IStationPlaylist і створює PlaybackQueueIterator для обходу плейлиста.
     - Visitor: ListeningStatsVisitor реалізує логіку збору статистики, а сутності (Track, DjStream) надають метод Accept().
   - Усі інтерфейси (IAudioProcessor, IRepository, IStationService) знаходяться тут, змушуючи інші шари залежати від абстракцій.
3. RadioStation.Data (Data Layer / Серверний Шар Даних):
   - Цей шар відповідає виключно за збереження та отримання даних з MS SQL Server.
   - Він містить реалізації інтерфейсів репозиторіїв (TrackRepository, StationRepository тощо), визначених у Domain.
   - Центральним елементом є ApplicationContext (контекст Entity Framework Core).

Взаємодія "Оживлених" Патернів

Діаграма ілюструє "повноцінні механізми" системи:

- Сценарій "Завантаження треку":

1. DjController (Клієнт) отримує запит.
2. Він викликає IAudioProcessor (Фасад).
3. AudioProcessingFacade (Фасад) використовує StreamFactory (Фабрику) для створення IAudioStream (Продукту).
4. "Продукт" (напр., LowBitrateStream) використовує IAudioConverter (Адаптер) для конвертації файлу.
5. Наприкінці AudioProcessingFacade використовує ITrackRepository для збереження результату в ApplicationContext (БД).

- Сценарій "Прослуховування":
  1. HomeController (Клієнт) отримує RadioStationEntity від IStationService.
  2. Listen.cshtml (Клієнт) викликає model.CreateIterator().
  3. RadioStationEntity (який реалізує IStationPlaylist) створює PlaybackQueueIterator (Ітератор) для обходу свого плейлиста.
- Сценарій "Статистика":
  1. AdminController (Клієнт) завантажує дані через репозиторії.
  2. Він створює ListeningStatsVisitor (Відвідувача).
  3. Він "проганяє" Відвідувача по об'єктах (Track, DjStream), викликаючи їхні методи Accept().

Механізм завантаження треку

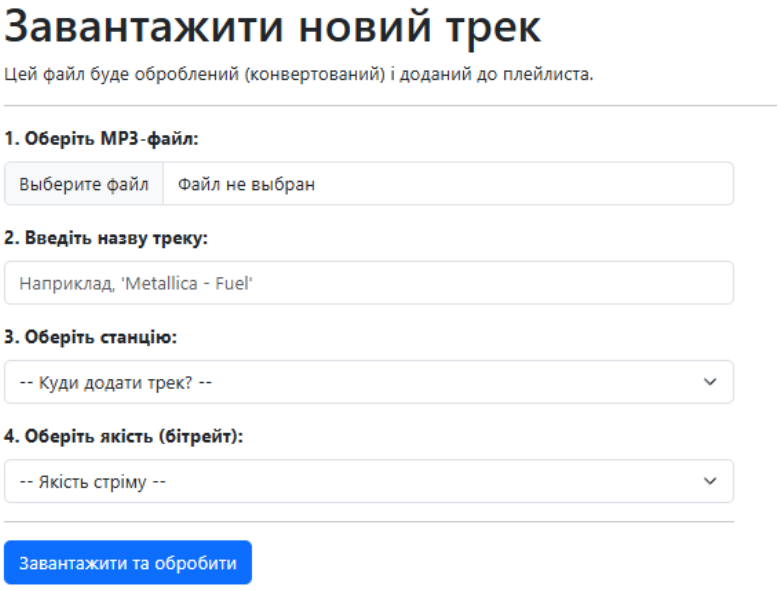

Рис. 3 - Upload.cshtml у браузері

Клієнтська частина: це показує Клієнта — інтерфейс, де користувач (Діджей) вводить дані (файл, назву, бітрейт). Це і є той клієнт, який готує запит на сервер.

```
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<IActionResult> Upload(IFormFile trackFile, string title, Guid stationId, int bitrate)
{
    if (trackFile == null || trackFile.Length == 0)
    {
        ModelState.AddModelError("trackFile", "Будь ласка, оберіть MP3-файл.");
    }
    if (string.IsNullOrEmpty(title))
    {
        ModelState.AddModelError("title", "Будь ласка, введіть назву треку.");
    }
    if (stationId == Guid.Empty)
    {
        ModelState.AddModelError("stationId", "Будь ласка, оберіть станцію.");
    }
    if (bitrate == 0)
    {
        ModelState.AddModelError("bitrate", "Будь ласка, оберіть якість (бітрейт).");
    }

    if (!ModelState.IsValid)
    {
        var stations = await _stationService.GetAllStationsAsync();
        ViewBag.Stations = new SelectList(stations, "StationId", "StationName");
        return View();
    }
```

Рис. 4 – DjController, а саме [HttpPost] Upload(...)

Серверна частина: Сервер у момент прийому HTTP-запиту від Клієнта. Тут видно, як дані з HTML-форми перетворюються на C# параметри.

```
try
{
    await _audioProcessor.ProcessNewTrackAsync(
        tempFilePath: tempFilePath,
        title: title,
        stationId: stationId,
        djId: djUser.UserId,
        bitrate: bitrate
    );
}
```

Рис. 5 – DjController, виклик фасаду

Middleware: зв'язок між Сервером та Middleware. Контролер не знає про FFmpeg чи репозиторії, він просто передає роботу бізнес-логіці.

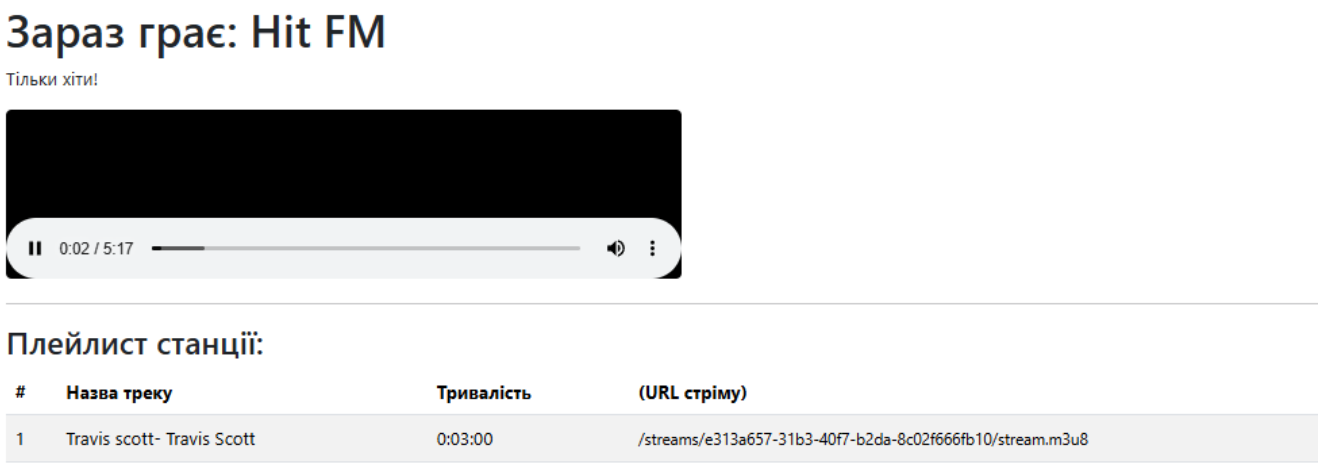

Рис. 6 – Результат: трек завантажено та запущено

Паттерни:

Iterator: обхід плейлиста станції на сторінці прослуховування.

```
public IPlaylistIterator CreateIterator()
{
    var tracks = this.Playbacks
        .OrderBy(p => p.QueuePosition)
        .Select(p => p.Track)
        .ToList();
    return new PlaybackQueueIterator(tracks);
}
```

Рис. 7 – Метод Iterator в RadioStationEntity.cs

```
@while (!iterator.IsDone())
{
    var Track? track = iterator.Current();
    <tr>
        <td>@(position++)</td>
        <td>@track.Title</td>
        <td>@track.Duration.ToString(format: "g")</td>
        <td><small>@track.HlsUrl</small></td> @* Показуємо URL для відладки *@
    </tr>
    iterator.Next();
}
```

Рис. 8 – Цикл @while (!iterator.IsDone()) в Listen.cshtml

	QueueId	TrackId	StationId	AddedById	QueuePosition
1	3A0F04C4-2BAF-4EB4-8F9E-EB201FE7BA89	9AC3F17D-D451-4DA5-9E39-BF12ABB28253	CFE8E492-4B58-4138-B29C-57D50693BA28	CC39E904-D61D-40FB-B2EC-6F7E4411D560	1

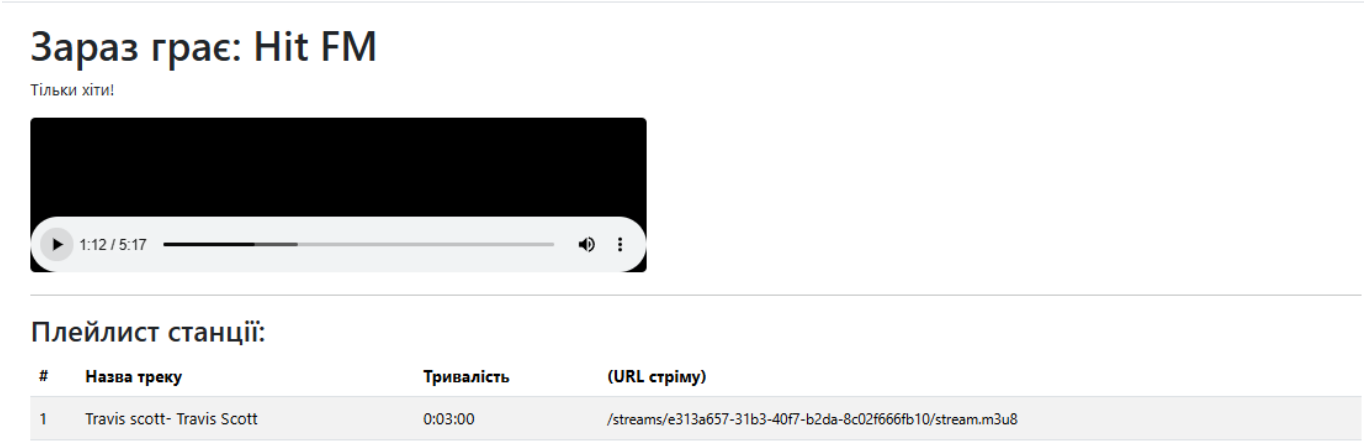

Рис. 9 – Відображення треків з БД

Façade + Factory Method + Adapter: патерни тепер працюють разом у "повноцінному механізмі" завантаження треку Діджеєм.

Facade (Фасад): менеджер для DjController. Він приховує складність процесу (вибір бітрейту, виклик Фабрики, виклик Адаптера, збереження в таблиці БД) за одним простим методом: ProcessNewTrackAsync().

```
try
{
    await _audioProcessor.ProcessNewTrackAsync(
        tempFilePath: tempFilePath,
        title: title,
        stationId: stationId,
        djId: djUser.UserId,
        bitrate: bitrate
    );
}
catch (Exception ex)
```

Рис. 10 – DjController.cs, виклик _audioProcessor.ProcessNewTrackAsync(...)

Factory Method: створити правильний продукт (LowBitrateStream або HighBitrateStream) на основі вибору Діджея ("Якість").

```
public async Task ProcessNewTrackAsync(string tempFilePath, string title, Guid stationId, Guid djId, int bitrate)
{
    Console.WriteLine($"[Facade] Починаю обробку: {title} з бітрейтом {bitrate}kb/s");

    var streamProduct = _streamFactory.Create(bitrate);
    Console.WriteLine($"[Facade] Викликаю {streamProduct.GetType().Name} (який викличе Adapter)...");

    var hlsWebUrl = await streamProduct.CreateStreamAsync(tempFilePath);
    Console.WriteLine($"[Facade] Adapter завершив роботу. HLS створено: {hlsWebUrl}");

    var hardcodedDuration = TimeSpan.FromMinutes(3);
    var newTrack = new Track
    {
        TrackId = Guid.NewGuid(),
        Title = title,
        Duration = hardcodedDuration,
        UploadedById = djId,
        HlsUrl = hlsWebUrl
    };
    _trackRepository.AddEntity(newTrack);
    Console.WriteLine($"[Facade] Додаю {newTrack.Title} в таблицю Tracks.");

    var nextPosition = _queueRepository.GetAll()
        .Count(q => q.StationId == stationId) + 1;
    var newQueueEntry = new PlaybackQueue
    {
        QueueId = Guid.NewGuid(),
        TrackId = newTrack.TrackId,
```

Рис. 11 – DjController.cs, виклик _streamFactory.Create(bitrate)

Adapter: використовують продукти, створені Фабрикою. Він бере C#-команду (наприклад, "конвертуй цей файл з бітрейтом 64k") і "адаптує" її у команду, зрозумілу для зовнішньої програми (ffmpeg.exe).

```
public class LowBitrateStream : IAudioStream
{
    2 references
    private readonly IAudioConverter _converter;
    2 references
    private const int Bitrate = 64;

    1 reference
    public LowBitrateStream(IAudioConverter converter)
    {
        _converter = converter;
    }

    1 reference
    public int GetBitrate() => Bitrate;

    2 references
    public async Task<string> CreateStreamAsync(string inputAudioPath)
    {
        return await _converter.ConvertToHlsAsync(inputAudioPath, Bitrate);
    }
```

Рис. 12 – Метод CreateStreamAsync, де "продукт" (створений Фабрикою) викликає _converter.ConvertToHlsAsync(...)

```
[Facade] Починаю обробку: Sicko Mode (minus) з бітрейтом 128kb/s
[Facade] Викликаю StandardBitrateStream (який викличе Adapter)...
FFMPEG LOG: ffmpeg version 2025-10-30-git-00c23bafb0-essentials_build-www.gyan.dev Copyright (c) 2000-2025 the FFmpeg developers
  built with gcc 15.2.0 (Rev8, Built by MSYS2 project)
  configuration: --enable-gpl --enable-version3 --enable-static --disable-w32threads --disable-autodetect --enable-cairo --enable-fontconfig --enabl
e-iconv --enable-gnutls --enable-libxml2 --enable-gmp --enable-bzlib --enable-lzma --enable-zlib --enable-libsrt --enable-libssh --enable-libzmq --e
nable-avisynth --enable-sdl2 --enable-libwebp --enable-libx264 --enable-libx265 --enable-libxvid --enable-libaom --enable-libopenjpeg --enable-libvp
x --enable-mediafoundation --enable-libass --enable-libfreetype --enable-libfribidi --enable-libharfbuzz --enable-libvidstab --enable-libvmaf --enab
le-libzimg --enable-amf --enable-cuda-llvm --enable-cuvid --enable-dxva2 --enable-d3d11va --enable-d3d12va --enable-ffnvcodec --enable-libvpl --enab
le-nvdec --enable-nvenc --enable-vaapi --enable-openal --enable-libgme --enable-libopenmpt --enable-libopencore-amrwb --enable-libmp3lame --enable-l
ibtheora --enable-libvo-amrwbenc --enable-libgsm --enable-libopencore-amrnb --enable-libopus --enable-libspeex --enable-libvorbis --enable-librubber
band
  libavutil      60. 16.100 / 60. 16.100
  libavcodec     62. 16.100 / 62. 16.100
  libavformat    62.  6.101 / 62.  6.101
  libavdevice    62.  2.100 / 62.  2.100
  libavfilter    11.  9.100 / 11.  9.100
  libswscale      9.  3.100 /  9.  3.100
  libswresample   6.  2.100 /  6.  2.100
Input #0, mp3, from 'C:\Users\38093\AppData\Local\Temp\0210f4c1-eb79-421f-a68d-8c01b68ac979.mp3':
  Duration: 00:05:17.32, start: 0.025057, bitrate: 320 kb/s
  Stream #0:0: Audio: mp3 (mp3float), 44100 Hz, stereo, fltp, 320 kb/s, start 0.025057
    Metadata:
      encoder         : LAME3.100
    Side data:
      Replay Gain: track gain - -7.800000, track peak - unknown, album gain - unknown, album peak - unknown,
Stream mapping:
  Stream #0:0 -> #0:0 (mp3 (mp3float) -> aac (native))
Press [q] to stop, [?] for help
[mpegts @ 0000011c5a639a40] frame size not set
Output #0, hls, to 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac979\stream.m3u8'
:
  Metadata:
    encoder         : Lavf62.6.101
  Stream #0:0: Audio: aac (LC), 44100 Hz, stereo, fltp, 128 kb/s
    Metadata:
      encoder         : Lavc62.16.100 aac
    Side data:
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
9\stream.m3u8.tmp' for writing
size=N/A time=00:00:19.82 bitrate=N/A speed=39.3x elapsed=0:00:00.50
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
9/seg001.ts' for writing
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
9\stream.m3u8.tmp' for writing
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
9/seg002.ts' for writing
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
9\stream.m3u8.tmp' for writing
size=N/A time=00:00:39.89 bitrate=N/A speed=38.7x elapsed=0:00:01.03
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
9/seg003.ts' for writing
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
9\stream.m3u8.tmp' for writing
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
9/seg004.ts' for writing
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
9\stream.m3u8.tmp' for writing
size=N/A time=00:00:55.40 bitrate=N/A speed=35.9x elapsed=0:00:01.54
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
9/seg005.ts' for writing
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
9\stream.m3u8.tmp' for writing
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
9/seg006.ts' for writing
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
9\stream.m3u8.tmp' for writing
size=N/A time=00:01:11.26 bitrate=N/A speed=34.6x elapsed=0:00:02.05
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
9/seg007.ts' for writing
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
9\stream.m3u8.tmp' for writing
size=N/A time=00:01:23.42 bitrate=N/A speed=32.4x elapsed=0:00:02.57
[hls @ 0000011c5a5bfcc0] Opening 'C:\MyProjects\RadioStationSolution\RadioStationSolution.WebApp\wwwroot\streams\0210f4c1-eb79-421f-a68d-8c01b68ac97
```

Рис. 13 Виконання процесу

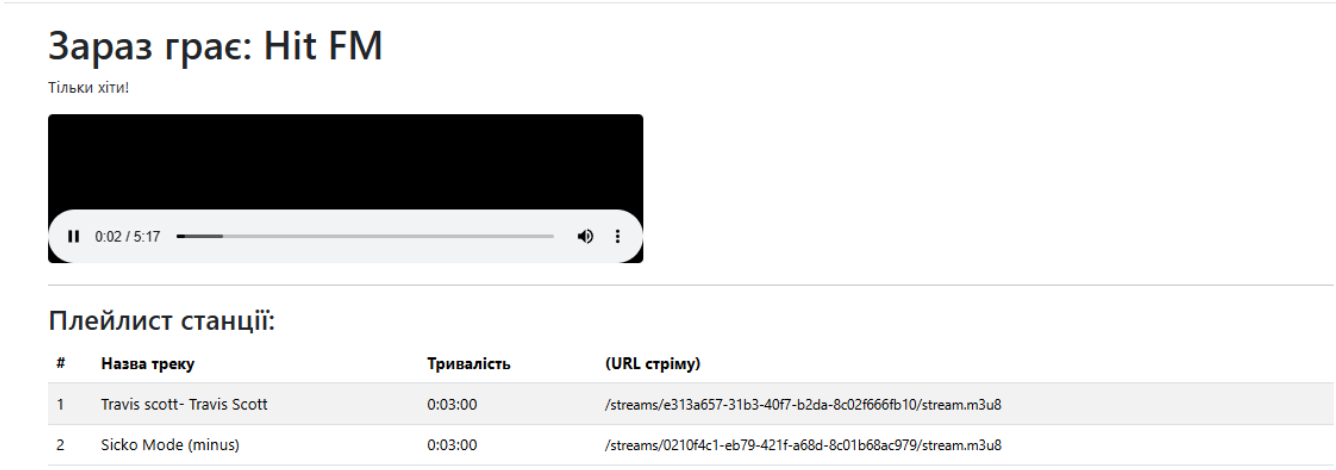

Рис. 14 Результат

Visitor: Збір статистики (TotalMinutes, TotalTracks...) з реальних даних у базі.

```
public async Task<IActionResult> ShowStats()
{
    var visitor = new ListeningStatsVisitor();
    var allTracks = await _trackRepo.GetAll().ToListAsync();
    var allStreams = await _streamRepo.GetAll().ToListAsync();
    var allQueueItems = await _queueRepo.GetAll().ToListAsync();

    foreach (var track in allTracks)
    {
        track.Accept(visitor);
    }

    foreach (var stream in allStreams)
    {
        stream.Accept(visitor);
    }

    foreach (var item in allQueueItems)
    {
        item.Accept(visitor);
    }

    return View(visitor);
}
```

Рис. 15 ShowStats() з AdminController.cs

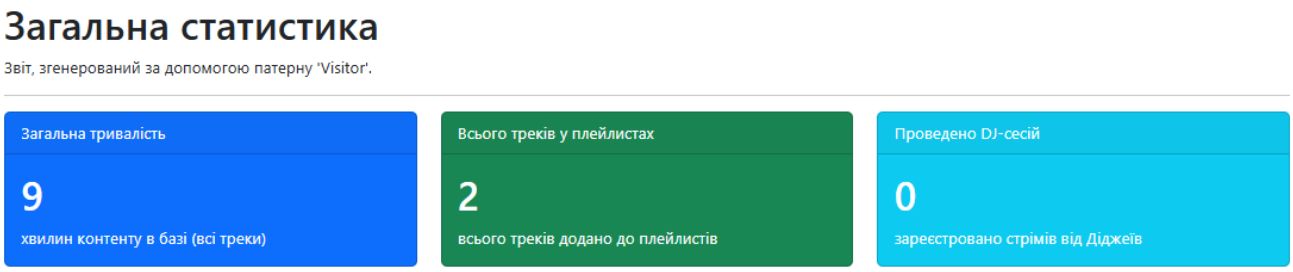

Рис. 15 Скріншот сторінки, де відображаються картки зі статистикою

4. Висновок

У ході виконання лабораторної роботи було детально вивчено та практично реалізовано клієнт-серверну архітектуру (Client-Server) на базі проєкту "Online Radio Station". Було досягнуто чіткого розділення відповідальності між трьома основними компонентами системи. Серверна частина , побудована на ASP.NET Core , виконує роль основного обчислювального центру: обробляє HTTP-запити та реалізує всю складну бізнес-логіку, таку як конвертація аудіо (патерни Facade, Adapter, Factory Method) та збір статистики (патерн Visitor). Клієнтська частина реалізована у вигляді "тонкого клієнта" на Razor-сторінках (.cshtml) , який відповідає лише за візуалізацію (форми завантаження , плеєр hls.js , відображення плейлистів) та відправку запитів на сервер. Посередницький шар (Middleware) (проекти RadioStation.Domain та RadioStation.Services) успішно інкапсулював усі контракти (інтерфейси патернів) та моделі даних, забезпечуючи слабке зв'язування та інверсію залежностей. Результатом роботи є повноцінний, "оживлений" веб-додаток, який демонструє практичну реалізацію всіх необхідних патернів проектування в рамках сучасної, розподіленої клієнт-серверної системи.

5. Контрольні питання

1. Що таке клієнт-серверна архітектура?

Клієнт-серверна архітектура — це модель організації комп’ютерних систем, де клієнти (зазвичай користувацькі програми або пристрої) роблять запити на сервер, який обробляє ці запити і повертає результати. Клієнт відповідає за інтерфейс користувача та ініціацію запитів, сервер — за обробку даних, зберігання та логіку.

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

SOA — це архітектурний підхід, де функціональність програми реалізована у вигляді сервісів — автономних компонентів, що виконують конкретні бізнесзавдання. Кожен сервіс має чіткий інтерфейс і взаємодіє з іншими через стандартизовані протоколи (наприклад, HTTP, SOAP, REST).

3. Якими принципами керується SOA?

Основні принципи SOA:

• Автономність сервісів — сервіси працюють незалежно.

• Стандартизовані інтерфейси — сервіси взаємодіють через визначені API.

• Повторне використання — сервіси можна використовувати в різних системах.

• Легко інтегрувати — сервіси повинні легко поєднуватись у складні процеси.

• Слабке зв’язування (loose coupling) — зміни в одному сервісі мінімально впливають на інші.

4. Як між собою взаємодіють сервіси в SOA?

Сервіси взаємодіють через стандартизовані повідомлення або API, наприклад через SOAP або REST. Кожен сервіс публікує свій інтерфейс (WSDL, OpenAPI), і інші сервіси можуть викликати його методи, обмінюючись даними у формі XML, JSON або інших форматах.

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

• Реєстр сервісів (Service Registry) — централізована база, де зареєстровані всі сервіси та їхні інтерфейси.

• Документація API (наприклад OpenAPI/Swagger).

• Запити здійснюються через стандартні протоколи (HTTP, SOAP, REST) за адресою сервісу і з використанням описаних методів та форматів даних.

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

Переваги:

• Центральне зберігання даних, легший контроль безпеки.

• Легко масштабувати сервери.

• Клієнти можуть бути простими, вся логіка на сервері.

Недоліки:

• Сервер може стати вузьким місцем (single point of failure).

• Високі вимоги до потужності сервера при великій кількості клієнтів.

• Залежність клієнта від сервера — без доступу до сервера система не працює.

7. У чому полягають переваги та недоліки однорангової (peer-to-peer) моделі взаємодії? Переваги:

• Відсутність централізованого сервера, підвищена стійкість до відмов.

• Можливість прямого обміну ресурсами між учасниками.

• Масштабування «горизонтальне» — додаючи вузли, підвищуєш потужність.

Недоліки:

• Складніше забезпечувати безпеку та контроль доступу.

• Кожен вузол відповідає за управління ресурсами.

• Важче координувати оновлення і синхронізацію даних.

8. Що таке мікросервісна архітектура? Мікросервісна архітектура — це підхід, де додаток складається з малих, незалежних сервісів, кожен з яких реалізує одну бізнес-функцію і може розгортатися окремо. Вона є розвитком SOA, але з більш дрібними і автономними компонентами.

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

• HTTP/HTTPS + REST

• gRPC

• SOAP

• Message brokers: RabbitMQ, Kafka, MQTT для асинхронної взаємодії

• WebSockets для реального часу

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

Це не повноцінна SOA, а локальне використання сервісів всередині одного додатку. SOA передбачає автономні, незалежні сервіси, доступні через стандартизовані протоколи для інших систем. Твій підхід — це архітектурний патерн «сервісний шар» (Service Layer), який організовує бізнес-логіку всередині одного проєкту.