**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**

**УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО**

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №3

по дисциплине: «Программирование сетевых приложений»

на тему: **«**Микро-сервисная архитектура**»**

Выполнил: студент гр. ИТИ-41

Ковшаров Г. Ю.

Принял: преподаватель

Гуменников Е.Д.

Гомель 2025

**Цель работы:** освоить методов построения приложений, основанных на микро-сервисной архитектуре. Научиться разбивать задачу на сервисы и выстраивать с ними взаимодействие.

**Задание:**

Разработать сетевое приложение согласно варианту. Необходимо реализовать *CRUD* операции для минимума трех сущностей и агрегатные вычисления минимум для пяти показателей (самый прибыльный месяц, самый популярный музыкальный альбом, длина маршрута, пройдённого водителем за период времени и т.п.) в соответствующей предметной области, **использовать микро-сервисы** в построении архитектуры системы. Для взаимодействия сервисов **использовать TCP или UDP сокеты**.

Таблица 1 – Вариант задания

|  |  |
| --- | --- |
| **Вариант** | **Предметная область** |
| 13 | Видеотека |

**Ход работы**

Для реализации системы видеопроката была разработана масштабируемая архитектура, построенная на принципе микросервисов, где каждый модуль выполняет специализированную задачу, обеспечивая гибкость и независимость компонентов. Такой дизайн позволил разделить ответственность между сервисами, упростить их обслуживание и дать возможность масштабировать каждый элемент отдельно в зависимости от нагрузки.

Система включает следующие ключевые компоненты:

Сервис видео (*VideoService*): отвечает за управление каталогом видеоконтента. Он предоставляет полный цикл операций *CRUD* для сущностей Video, включая добавление, обновление и удаление фильмов или сериалов. Данные хранятся в локальной базе данных *video.db* (*SQLite*), что гарантирует автономность сервиса.

Сервис клиентов (*CustomerService*): управляет информацией о пользователях системы. Этот модуль реализует CRUD-операции для сущностей Customer, обеспечивая регистрацию, редактирование и удаление профилей. Он опирается на собственное хранилище *customer*.*db* и функционирует независимо от других сервисов.

Сервис аренды (*RentalService*): служит центральным узлом для обработки операций аренды. Он управляет сущностями *Rental* и *Payment*, поддерживая создание, обновление и удаление записей об арендах, а также обработку платежей. Для проверки доступности видео и клиентов сервис обращается к *VideoService* и *CustomerService*, демонстрируя межсервисное взаимодействие.

Сервис отчетов (*ReportingService*): предназначен для аналитики и агрегации данных. Без собственной базы данных, он собирает информацию из *RentalService*, вычисляя такие метрики, как общая выручка, самый популярный и прибыльный видеоконтент, средняя длительность аренды и топ-клиент по затратам.

*API* Шлюз (*ApiGateway*): выступает единственным входным каналом для всех внешних запросов. Клиентское приложение (*WPF*) взаимодействует только с шлюзом, который перенаправляет запросы к соответствующим микросервисам. Это обеспечивает скрытие внутренней структуры сети и упрощает добавление функций, таких как мониторинг или ограничения доступа.

Обмен данными между компонентами осуществляется через *TCP*-сокеты с использованием простого протокола, имитирующего *REST*-стиль. Запросы включают метод (*GET*, *POST* и т.д.), путь, заголовки (например, *Content*-*Length*) и тело в формате *JSON*. Каждый сервис развернут как отдельное консольное приложение с уникальным портом, а идентификаторы (*GUID*) применяются для всех ключевых сущностей (*Video*, *Customer*, *Rental*), что минимизирует риск коллизий в распределённой системе. Пример логов работы шлюза можно увидеть на рисунке 1.

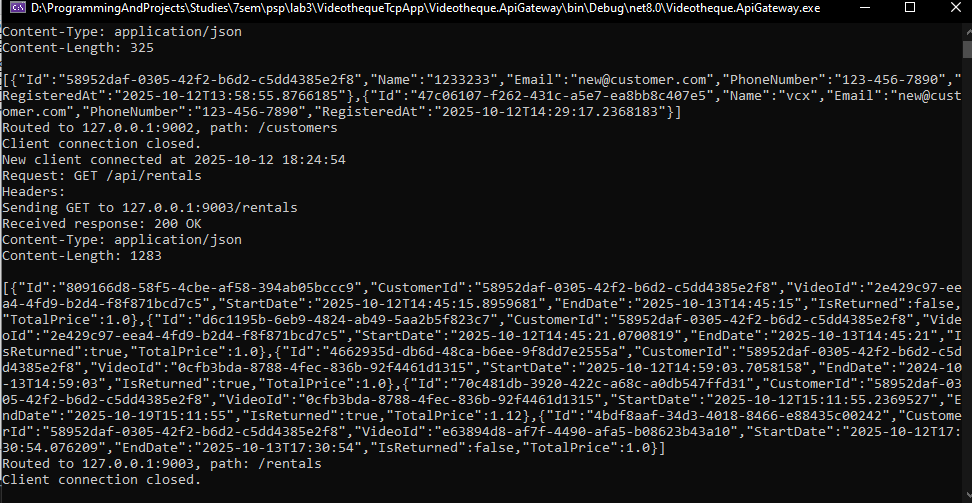


Рисунок 1 – Логи работы *API* *Gateway*

Клиентская часть реализована как десктопное приложение на базе *WPF* с интуитивно понятным интерфейсом, разделённым на вкладки для разных функций:

Аренда видео: позволяет выбирать фильмы из каталога и оформлять аренду с указанием даты возврата.

Управление каталогом: предоставляет возможность просматривать, добавлять, изменять и удалять видеоконтент.

Управление клиентами: включает просмотр списка пользователей, их редактирование и удаление.

Аналитика: отображает сгенерированные отчёты на основе данных от *ReportingService*.

Сетевая логика вынесена в отдельную библиотеку, что обеспечивает чистоту кода интерфейса и его концентрацию на визуализации. Все операции выполняются асинхронно, что сохраняет отзывчивость приложения даже при интенсивной работе с сервером. Скриншоты интерфейса можно посмотреть на следующих рисунках.

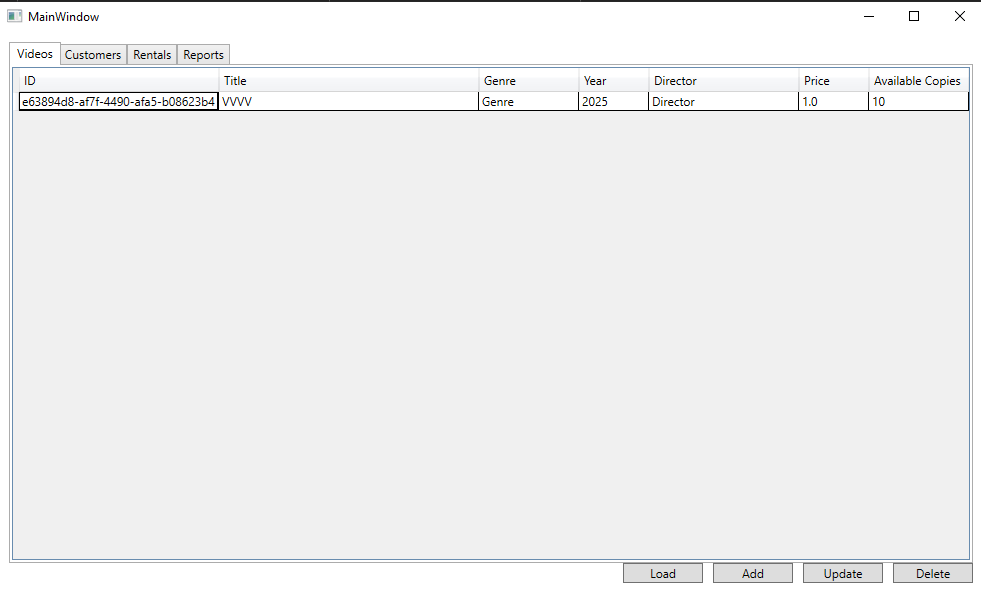


Рисунок 2 – Вкладка *Video*

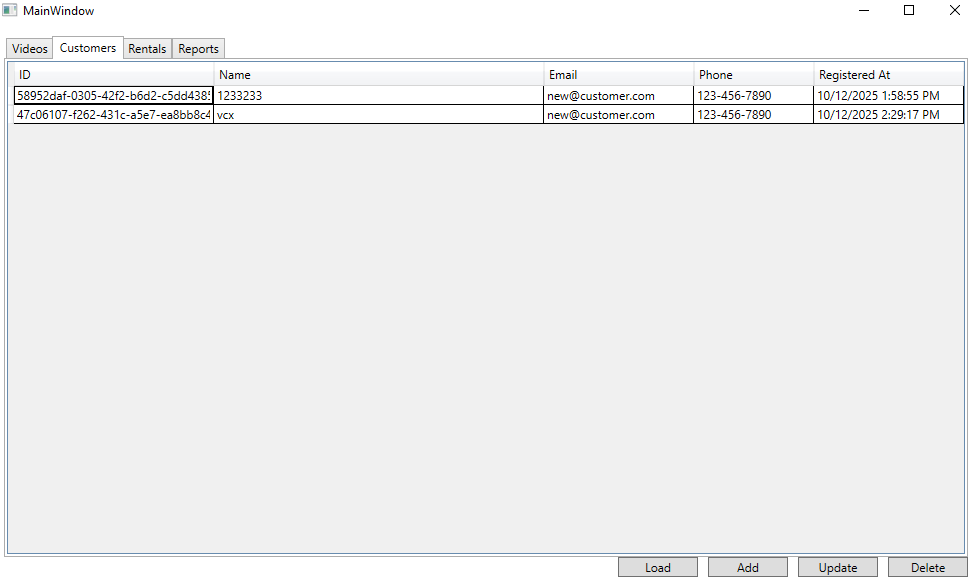


Рисунок 3 – Вкладка *Customers*

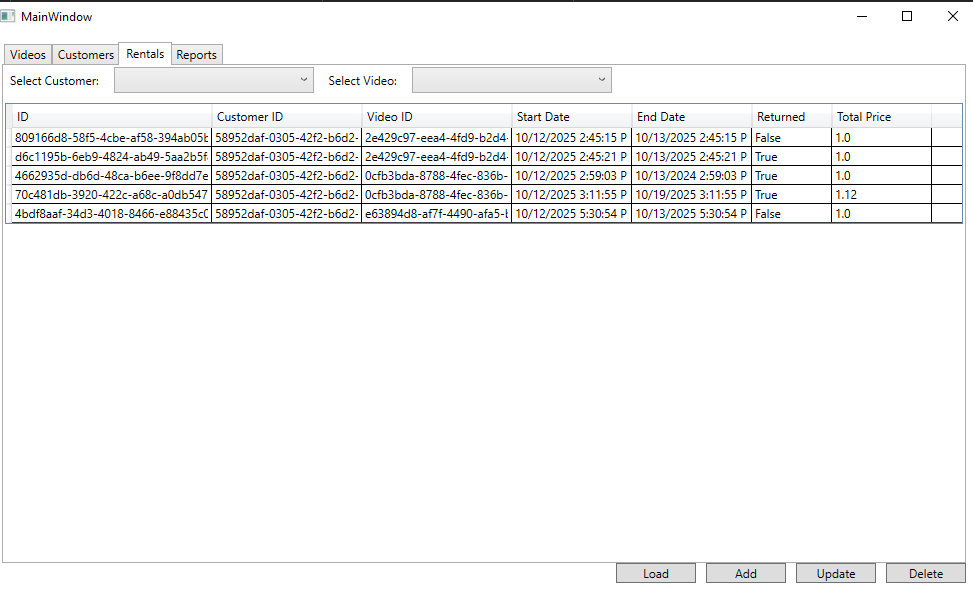


Рисунок 4 – Вкладка *Rentals*

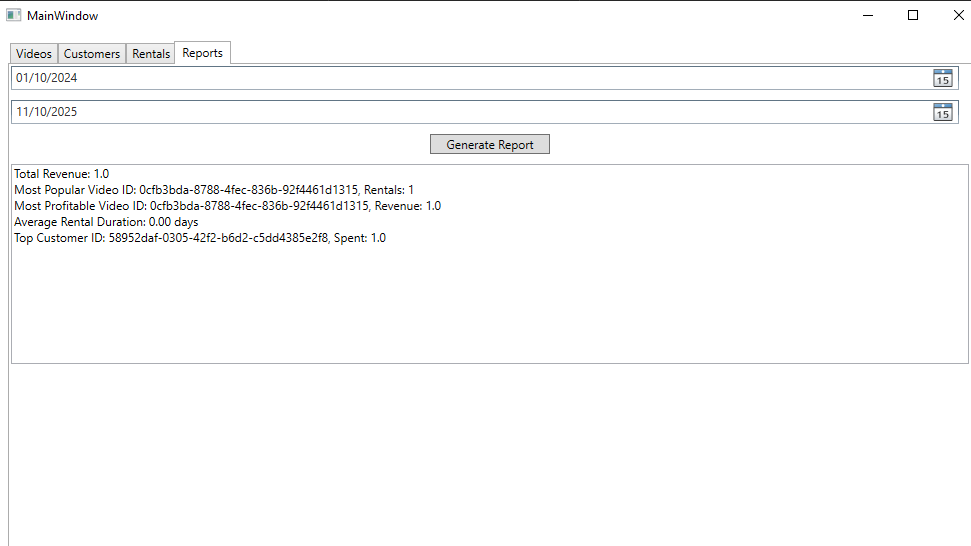


Рисунок 4 – Вкладка *Report*

Результатом работы стало создание надёжной распределённой системы, полностью удовлетворяющей поставленным задачам. Микросервисная архитектура обеспечила чёткое разделение ролей, изоляцию данных и удобство параллельной разработки. Все *CRUD*-операции для *Video*, *Customer* и *Rental* работают корректно, а *ReportingService* предоставляет точные аналитические данные. Система стабильно функционирует, сохраняя информацию в базах данных, а взаимодействие через *API* шлюз остаётся прозрачным для пользователей.

**Вывод:** в процессе выполнения лабораторной работы были освоены ключевые принципы и методы построения приложений на основе микросервисной архитектуры. Получен практический опыт декомпозиции сложной задачи на независимые сервисы, организации их сетевого взаимодействия с помощью *TCP*-сокетов и реализации единой точки входа через *API* шлюз. Созданное приложение является наглядным примером гибкой и масштабируемой системы, построенной по современным архитектурным стандартам.

**ПРИЛОЖЕНИЕ А**

(обязательное)

**Листинг программы**

MainWindow.cs  
using System.Collections.ObjectModel;

using System.Text;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Navigation;

using System.Windows.Shapes;

using VideothequeTcpApp.Models.Domain;

using VideothequeTcpApp.Networking;

namespace VideothequeTcpApp.Wpf

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

private readonly RestOverTcpClient \_client;

private readonly ObservableCollection<Video> \_videos = new();

public MainWindow()

{

InitializeComponent();

\_client = new RestOverTcpClient("127.0.0.1", 8888); // IP и порт сервера

VideosDataGrid.ItemsSource = \_videos;

}

private async void LoadButton\_Click(object sender, RoutedEventArgs e)

{

try

{

var all = await \_client.GetAsync<Video[]>("/videos");

\_videos.Clear();

foreach (var v in all ?? Array.Empty<Video>())

\_videos.Add(v);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

private async void AddButton\_Click(object sender, RoutedEventArgs e)

{

var video = new Video

{

Title = "New Video",

Genre = "Genre",

Year = 2025,

Director = "Director",

Price = 1.99m

};

try

{

var created = await \_client.PostAsync("/videos", video);

if (created != null) \_videos.Add(created);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

private async void UpdateButton\_Click(object sender, RoutedEventArgs e)

{

if (VideosDataGrid.SelectedItem is not Video selected) return;

try

{

var success = await \_client.PutAsync("/videos/" + selected.Id, selected);

if (success != null)

{

var index = \_videos.IndexOf(selected);

\_videos[index] = success;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

private async void DeleteButton\_Click(object sender, RoutedEventArgs e)

{

if (VideosDataGrid.SelectedItem is not Video selected) return;

try

{

await \_client.DeleteAsync("/videos/" + selected.Id);

\_videos.Remove(selected);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

}

}

MainWindow.xaml

<Window x:Class="VideothequeTcpApp.Wpf.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" xmlns:local="clr-namespace:VideothequeTcpApp.Wpf"

Title="MainWindow" Height="450" Width="800">

<Grid Margin="10">

<Grid.RowDefinitions>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<DataGrid x:Name="VideosDataGrid" AutoGenerateColumns="False" CanUserAddRows="False">

<DataGrid.Columns>

<DataGridTextColumn Header="ID" Binding="{Binding Id}" IsReadOnly="True" Width="200"/>

<DataGridTextColumn Header="Title" Binding="{Binding Title}" Width="\*"/>

<DataGridTextColumn Header="Genre" Binding="{Binding Genre}" Width="100"/>

<DataGridTextColumn Header="Year" Binding="{Binding Year}" Width="70"/>

<DataGridTextColumn Header="Director" Binding="{Binding Director}" Width="150"/>

<DataGridTextColumn Header="Price" Binding="{Binding Price}" Width="70"/>

</DataGrid.Columns>

</DataGrid>

<StackPanel Grid.Row="1" Orientation="Horizontal" HorizontalAlignment="Right">

<Button Content="Load" Width="80" Click="LoadButton\_Click" Margin="0,0,10,0"/>

<Button Content="Add" Width="80" Click="AddButton\_Click" Margin="0,0,10,0"/>

<Button Content="Update" Width="80" Click="UpdateButton\_Click" Margin="0,0,10,0"/>

<Button Content="Delete" Width="80" Click="DeleteButton\_Click"/>

</StackPanel>

</Grid>

</Window>

RestOverTcpClient.cs

using System.Net.Sockets;

using System.Text;

using System.Text.Json;

namespace VideothequeTcpApp.Networking

{

/// <summary>

/// Простая реализация HTTP-подобного клиента поверх TCP.

/// Поддерживает методы GET, POST, PUT и DELETE с JSON-телом запроса/ответа.

/// </summary>

public class RestOverTcpClient

{

private readonly string \_host; // Адрес сервера

private readonly int \_port; // Порт сервера

private readonly JsonSerializerOptions \_jsonOptions; // Опции сериализации JSON

public RestOverTcpClient(string host, int port)

{

\_host = host;

\_port = port;

\_jsonOptions = new JsonSerializerOptions

{

PropertyNameCaseInsensitive = true // Игнорируем регистр при десериализации

};

}

// Обёртки для отправки запросов разных типов

public Task<T?> GetAsync<T>(string path) => \_sendRequestAsync<T>("GET", path);

public Task<T?> PostAsync<T>(string path, T body) => \_sendRequestAsync<T>("POST", path, body);

public Task<T?> PutAsync<T>(string path, T body) => \_sendRequestAsync<T>("PUT", path, body);

public Task DeleteAsync(string path) => \_sendRequestAsync<object>("DELETE", path);

/// <summary>

/// Основной метод для отправки запроса на сервер.

/// Формирует HTTP-подобное сообщение и отправляет через TCP.

/// </summary>

private async Task<T?> \_sendRequestAsync<T>(string method, string path, object? bodyContent = null)

{

// Формируем стартовую строку и заголовки

var requestBuilder = new StringBuilder();

requestBuilder.AppendLine($"{method} {path}");

byte[]? bodyBytes = null;

if (bodyContent != null)

{

// Сериализация тела запроса в JSON

var jsonBody = JsonSerializer.Serialize(bodyContent);

bodyBytes = Encoding.UTF8.GetBytes(jsonBody);

requestBuilder.AppendLine("Content-Type: application/json");

requestBuilder.AppendLine($"Content-Length: {jsonBody.Length}");

}

requestBuilder.AppendLine(); // Пустая строка разделяет заголовки и тело

// Устанавливаем TCP-соединение

using var client = new TcpClient();

await client.ConnectAsync(\_host, \_port);

await using var stream = client.GetStream();

// Отправляем заголовки

var headerBytes = Encoding.UTF8.GetBytes(requestBuilder.ToString());

await stream.WriteAsync(headerBytes);

// Отправляем тело, если есть

if (bodyBytes != null)

await stream.WriteAsync(bodyBytes);

// Читаем весь ответ сервера

using var reader = new StreamReader(stream, Encoding.UTF8);

var rawResponse = await reader.ReadToEndAsync();

// Парсим и возвращаем результат

return \_parseResponse<T>(rawResponse);

}

/// <summary>

/// Разбор ответа сервера.

/// Проверяет статусный код и десериализует JSON-тело.

/// </summary>

private T? \_parseResponse<T>(string rawResponse)

{

if (string.IsNullOrWhiteSpace(rawResponse))

throw new Exception("Empty respond from server");

// Разделяем полученный ответ сервера на заголовки и тело

// HTTP-подобный формат: заголовки и тело отделяются пустой строкой (\r\n\r\n)

// Split с параметром 2 гарантирует, что мы разделим только на две части:

// 1) заголовки (headersPart)

// 2) тело запроса (bodyPart)

var parts = rawResponse.Split(new[] { "\r\n\r\n" }, 2, StringSplitOptions.None);

// Берём первую часть — это все заголовки сервера

var headersPart = parts[0];

// Если есть вторая часть, то это тело, иначе пустая строка

var bodyPart = parts.Length > 1 ? parts[1] : string.Empty;

// Первая строка заголовков содержит статус ответа сервера (например "200 OK")

// Для этого разделяем headersPart по символам конца строки (\r и \n) и берём первый элемент

var statusLine = headersPart.Split('\r', '\n')[0];

return statusLine switch

{

var s when s.StartsWith("200") || s.StartsWith("201") => JsonSerializer.Deserialize<T>(bodyPart, \_jsonOptions),

var s when s.StartsWith("204") => default,

\_ => throw new Exception($"Error: {statusLine}")

};

}

}

}

ServerHelper.cs

using System.Net.Sockets;

using System.Text;

using System.Text.Json;

using VideothequeTcpApp.Models.Domain;

using VideothequeTcpApp.VideoService.Application.Services.Implementations;

using VideothequeTcpApp.VideoService.Infrastructure.Persistence;

namespace VideothequeTcpApp.VideoService.Presentation.Helpers

{

public static class ServerHelper

{

// Настройки сериализации JSON, игнорируем регистр имен свойств

private static readonly JsonSerializerOptions \_jsonOptions = new JsonSerializerOptions

{

PropertyNameCaseInsensitive = true

};

/// <summary>

/// Метод для обработки одного подключения клиента к серверу.

/// Разбирает запрос, выполняет CRUD-операцию через сервис и отправляет ответ.

/// </summary>

public static void HandleClient(TcpClient client)

{

Console.WriteLine("New connection!");

// Получаем поток данных TCP-соединения

using var stream = client.GetStream();

// Поток для чтения текстовых данных из клиента

using var reader = new StreamReader(stream, Encoding.UTF8);

// Поток для записи текстового ответа клиенту

using var writer = new StreamWriter(stream, Encoding.UTF8) { AutoFlush = true };

AppDbContext? ctx = null; // Контекст базы данных для этого запроса

try

{

// -------------------

// 1) Читаем стартовую строку запроса (например "GET /videos")

// -------------------

var requestLine = reader.ReadLine();

if (string.IsNullOrEmpty(requestLine)) return; // Если строка пустая — выходим

Console.WriteLine(requestLine);

// Разделяем метод (GET, POST и т.д.) и путь (/videos, /videos/{id})

var parts = requestLine.Split(' ', 2);

var method = parts[0];

var path = parts.Length > 1 ? parts[1] : "/";

// -------------------

// 2) Читаем заголовки запроса

// -------------------

var headers = new Dictionary<string, string>(StringComparer.OrdinalIgnoreCase);

string? line;

while (!string.IsNullOrEmpty(line = reader.ReadLine()))

{

var headerParts = line.Split(':', 2);

if (headerParts.Length == 2)

headers[headerParts[0].Trim()] = headerParts[1].Trim(); // Добавляем заголовок в словарь

}

// -------------------

// 3) Читаем тело запроса (если есть Content-Length)

// -------------------

string? body = null;

if (headers.TryGetValue("Content-Length", out var contentLengthValue) &&

int.TryParse(contentLengthValue, out var contentLength) && contentLength > 0)

{

var buffer = new char[contentLength];

var read = 0;

// Считываем ровно Content-Length символов из потока

while (read < contentLength)

{

var r = reader.ReadBlock(buffer, read, contentLength - read);

if (r == 0) break; // Конец потока

read += r;

}

body = new string(buffer, 0, read);

}

// -------------------

// 4) Создаём контекст базы данных и сервис для выполнения операций

// DbContext не потокобезопасен — создаём новый на каждый запрос

// имитируем SCOPED лайфтайм

// -------------------

ctx = new AppDbContextFactory().CreateDbContext(Array.Empty<string>());

var videosService = new VideosService(ctx);

string response;

// -------------------

// 5) Обрабатываем запрос в зависимости от метода и пути

// -------------------

// GET /videos — вернуть все видео

if (method == "GET" && path == "/videos")

{

var all = videosService.GetAll();

response = CreateResponse("200 OK", all);

}

// GET /videos/{id} — вернуть конкретное видео

else if (method == "GET" && path.StartsWith("/videos/"))

{

var idPart = path.Substring("/videos/".Length);

if (!Guid.TryParse(idPart, out var id))

{

response = CreateResponse("400 Bad Request");

}

else

{

var video = videosService.GetById(id);

response = video != null ? CreateResponse("200 OK", video) : CreateResponse("404 Not Found");

}

}

// POST /videos — создать новое видео

else if (method == "POST" && path == "/videos")

{

if (string.IsNullOrEmpty(body))

{

response = CreateResponse("400 Bad Request");

}

else

{

var dto = JsonSerializer.Deserialize<Video>(body, \_jsonOptions);

if (dto == null)

response = CreateResponse("400 Bad Request");

else

{

var created = videosService.Create(dto);

Console.WriteLine($"Video added: {created.Title}");

response = CreateResponse("201 Created", created);

}

}

}

// PUT /videos/{id} — обновить существующее видео

else if (method == "PUT" && path.StartsWith("/videos/"))

{

var idPart = path.Substring("/videos/".Length);

if (!Guid.TryParse(idPart, out var id))

{

response = CreateResponse("400 Bad Request");

}

else if (string.IsNullOrEmpty(body))

{

response = CreateResponse("400 Bad Request");

}

else

{

var dto = JsonSerializer.Deserialize<Video>(body, \_jsonOptions);

if (dto == null)

response = CreateResponse("400 Bad Request");

else

{

// Устанавливаем ID из URL и обновляем запись

dto.Id = id;

var ok = videosService.Update(dto);

if (ok)

{

var updated = videosService.GetById(id);

response = CreateResponse("200 OK", updated);

Console.WriteLine($"Video updated, ID: {id}");

}

else response = CreateResponse("404 Not Found");

}

}

}

// DELETE /videos/{id} — удалить видео

else if (method == "DELETE" && path.StartsWith("/videos/"))

{

var idPart = path.Substring("/videos/".Length);

if (!Guid.TryParse(idPart, out var id))

{

response = CreateResponse("400 Bad Request");

}

else

{

var success = videosService.Delete(id);

if (success)

{

Console.WriteLine($"Video deleted, ID: {id}");

response = CreateResponse("204 No Content");

}

else response = CreateResponse("404 Not Found");

}

}

else

{

// Некорректный метод или путь

response = CreateResponse("400 Bad Request");

}

// Отправляем сформированный ответ клиенту

writer.Write(response);

}

catch (Exception ex)

{

// Логируем ошибку и отправляем код 500

Console.WriteLine($"Error: {ex.Message}");

writer.Write(CreateResponse("500 Internal Server Error"));

}

finally

{

// Закрываем DbContext и соединение

ctx?.Dispose();

client.Close();

Console.WriteLine("Conntection closed.");

}

}

// Создаёт ответ с телом в формате JSON

private static string CreateResponse<T>(string status, T body)

{

var jsonBody = JsonSerializer.Serialize(body, \_jsonOptions);

return $"{status}\r\nContent-Type: application/json\r\n\r\n{jsonBody}";

}

// Создаёт ответ без тела (например 204 No Content)

private static string CreateResponse(string status) => $"{status}\r\n\r\n";

}

}

VideosService.cs

using VideothequeTcpApp.Models.Domain;

using VideothequeTcpApp.VideoService.Application.Services.Interfaces;

using VideothequeTcpApp.VideoService.Infrastructure.Persistence;

namespace VideothequeTcpApp.VideoService.Application.Services.Implementations

{

public class VideosService : IVideosService

{

private readonly AppDbContext \_context;

// Конструктор с внедрением зависимостей DbContext

public VideosService(AppDbContext context)

{

\_context = context;

}

// Параметрless конструктор, создающий новый DbContext через фабрику

// Полезен для ручного использования сервиса вне DI

public VideosService() : this(new AppDbContextFactory().CreateDbContext(Array.Empty<string>()))

{

}

// Получение всех видео

public IEnumerable<Video> GetAll()

{

return \_context.Videos.ToList(); // ToList() чтобы материализовать данные сразу

}

// Получение видео по ID

public Video? GetById(Guid id)

{

return \_context.Videos.FirstOrDefault(v => v.Id == id);

}

// Создание нового видео

public Video Create(Video video)

{

video.Id = Guid.NewGuid(); // Генерация уникального ID

\_context.Videos.Add(video); // Добавление в контекст

\_context.SaveChanges(); // Сохранение изменений в базе

return video;

}

// Обновление существующего видео

public bool Update(Video video)

{

var existing = \_context.Videos.FirstOrDefault(v => v.Id == video.Id);

if (existing == null) return false;

// Обновляем свойства

existing.Title = video.Title;

existing.Genre = video.Genre;

existing.Year = video.Year;

existing.Director = video.Director;

existing.Price = video.Price;

\_context.SaveChanges(); // Сохраняем изменения в базе

return true;

}

// Удаление видео по ID

public bool Delete(Guid id)

{

var existing = \_context.Videos.FirstOrDefault(v => v.Id == id);

if (existing == null) return false;

\_context.Videos.Remove(existing); // Удаление из контекста

\_context.SaveChanges(); // Сохраняем изменения в базе

return true;

}

}

}