

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ЛАБОРАТОРНА РОБОТА №9

ТЕМА: «Взаємодія компонентів системи.»

Виконала:

Студентка групи IA-34

Потейчук С.А.

Перевірив:

Мягкий М.Ю.

Зміст

Теоретичні відомості.....	3
Клієнт-серверна архітектура	3
Peer-to-Peer архітектура	3
Сервіс-орієнтована архітектура (SOA)	4
Мікросервісна архітектура	5
Хід роботи	6
Висновки	20
Контрольні питання	21

Теоретичні відомості

Клієнт-серверна архітектура

Клієнт-серверна архітектура є фундаментальною моделлю для розподілених систем, де чітко розмежовані ролі клієнта та сервера. Клієнти відповідають за представлення інформації та взаємодію з користувачем, тоді як сервери забезпечують зберігання, обробку даних та централізовану бізнес-логіку.

Існує спектр реалізацій клієнтської частини. Тонкі клієнти, як у традиційних веб-додатках, мінімізують локальну обробку, передаючи майже всі операції на сервер. Це спрощує розгортання та оновлення, оскільки зміни вносяться централізовано, але створює значне навантаження на серверну інфраструктуру. На протилежному кінці знаходяться товсті клієнти, такі як десктопні програми чи мобільні додатки, які виконують значну частину логіки локально. Це розвантажує сервери та дозволяє працювати в офлайн-режимі, але ускладнює процес оновлення. Проміжне положення займають односторінкові додатки (SPA), які поєднують переваги обох підходів: вони завантажуються з сервера, але потім виконують більшість операцій на клієнтській стороні через API, зменшуючи серверне навантаження та полегшуячи оновлення.

Типова трирівнева структура таких систем включає клієнтську частину для інтерфейсу користувача, проміжний шар із спільними компонентами та серверну частину з ядром бізнес-логіки. Така організація забезпечує модульність, масштабованість та ефективне розділення обов'язків.

Peer-to-Peer архітектура

Архітектура peer-to-peer (P2P) реалізує принципово інший підхід, заснований на повній децентралізації. У цій моделі кожен вузол мережі виступає одночасно і як клієнт, і як сервер, усуваючи необхідність у центральних координаційних точках.

Ключові принципи P2P включають повну децентралізацію, що підвищує стійкість системи до збоїв та атак, рівноправність усіх учасників мережі та ефективний розподіл ресурсів між вузлами. Ця архітектура знайшла широке застосування у різних сферах: файлообмінні системи на кшталт BitTorrent, криптовалютні платформи та блокчайн-технології, системи інтернет-телефонії та відеоконференцій, а також проекти розподілених обчислень.

Головними перевагами P2P є висока відмовостійкість через відсутність єдиних точок відмови, здатність до масового масштабування та ефективне використання мережевих ресурсів. Однак ця архітектура має і суттєві виклики, включаючи складність забезпечення безпеки в умовах децентралізації, проблеми синхронізації даних між численними вузлами, зниження ефективності пошуку ресурсів при зростанні мережі, а також юридичні складнощі контролю за розповсюдженням контенту.

Вибір між клієнт-серверною та P2P архітектурою визначається конкретними вимогами проекту щодо контролю, масштабованості, безпеки та надійності, оскільки кожен підхід має свої унікальні переваги та обмеження.

Сервіс-орієнтована архітектура (SOA)

Сервіс-орієнтована архітектура (SOA) представляє собою модульний підхід до розробки програмного забезпечення, заснований на використанні розподілених, слабко пов'язаних сервісів з стандартизованими інтерфейсами взаємодії. Ця архітектура виникла як еволюційна альтернатива монолітним системам, де вся функціональність була зосереджена в єдиному програмному комплексі.

В основі SOA лежить концепція веб-сервісів, які взаємодіють через стандартні протоколи, такі як HTTP з використанням SOAP або REST. Кожен сервіс інкапсулює конкретну бізнес-функцію, наприклад, управління запасами на складі або обробку платежів, і надає її через чітко визначений інтерфейс. Ключовою особливістю SOA є те, що сервіси взаємодіють виключно через обмін повідомленнями, без прямого доступу до спільних ресурсів, таких як бази даних.

Важливим аспектом SOA є реєстрація сервісів у спеціальних реєстрах, що дозволяє командам розробників легко знаходити та використовувати наявні сервіси. Часто для оркестрації взаємодії між сервісами використовується шина корпоративних сервісів (Enterprise Service Bus), яка виступає централізованим компонентом для обміну даними. SOA також дозволяє інтегрувати застарілі системи шляхом створення спеціальних обгорток, що значно знижує вартість модернізації та спрощує поступовий перехід до нової архітектури.

Мікросервісна архітектура

Мікросервісна архітектура є природним розвитком сервіс-орієнтованого підходу, що робить акцент на створенні серверних додатків як набору дрібних, автономних служб. Кожен мікросервіс є компонентом з чітко визначеними межами, який можна розгорнати незалежно та який взаємодіє з іншими сервісами через легкі протоколи зв'язку на основі повідомлень.

Відповідно до визначення, запропонованого в книзі "Архітектура мікросервісів", мікросервіс представляє собою автономний компонент, орієнтований на певні функціональні можливості в межах конкретного обмеженого контексту предметної області. Кожен сервіс функціонує у власному процесі та має повний життєвий цикл розробки, тестування та розгортання.

Основною перевагою мікросервісної архітектури є здатність забезпечувати ефективне супроводження складних систем з високими вимогами до масштабованості. Завдяки розподілу функціональності на незалежні компоненти, система стає більш гнучкою та стійкою до змін. Кожен мікросервіс може розвиватися власним шляхом, використовувати оптимальні технології для своїх завдань та масштабуватися незалежно від інших компонентів системи.

Цей підхід особливо ефективний для великих проектів з розподіленими командами розробників, оскільки дозволяє паралельно працювати над різними частинами системи без необхідності постійної координації та синхронізації змін.

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.

Хід роботи

7. Редактор зображень (state, prototype, memento, facade, composite, client server)

Редактор зображень має такі функціональні можливості: відкриття/збереження зображень у найпопулярніших форматах, застосування ефектів, наприклад поворот, розтягування, стиснення, кадрування зображення, можливість створення колажів шляхом «нашарування» зображень.

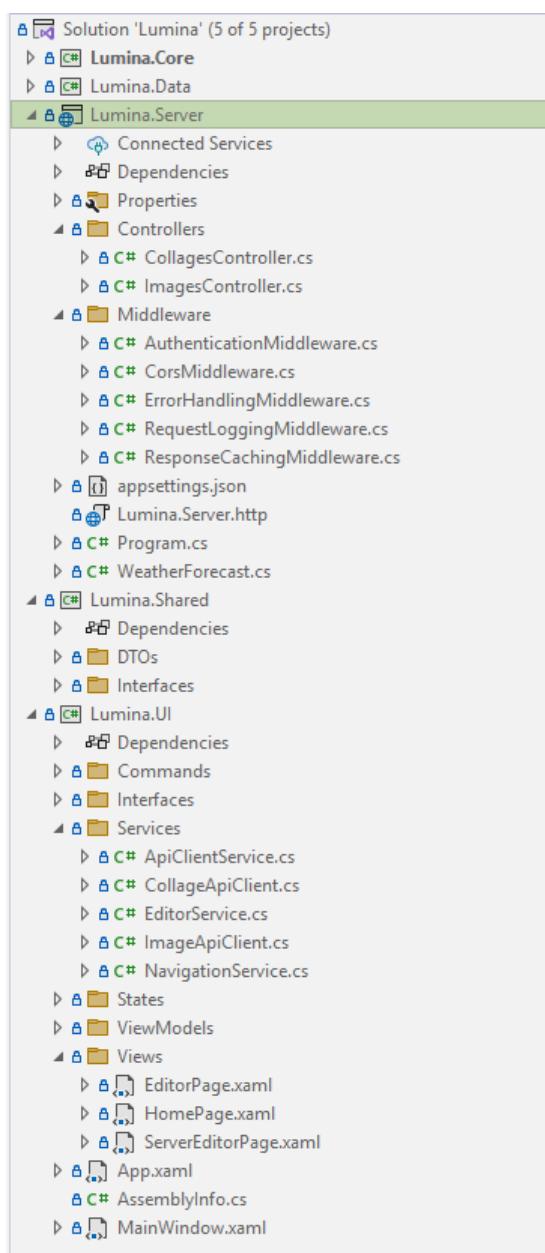


Рисунок 1 – Структура проєкту

Загальна структура проекту:

1. Lumina.Shared - Middleware (Спільна бібліотека)
 - DTOs - Data Transfer Objects для передачі даних між клієнтом і сервером
 - Інтерфейси - Контракти API для обох сторін
2. Lumina.Server - Серверна частина (ASP.NET Core Web API)
 - ImagesController - управління зображеннями (upload, get, delete)
 - CollagesController - управління колажами (create, add layers, get, delete)
3. Lumina.UI - Клієнтська частина (WPF)
 - ApiClientService - головний клієнт для з'єднання з сервером
 - ImageApiClient - робота з зображеннями
 - CollageApiClient - робота з колажами

Middleware Pipeline:

- ErrorHandlingMiddleware - глобальна обробка помилок
- RequestLoggingMiddleware - логування всіх запитів
- CorsMiddleware - дозволяє cross-origin запити
- ResponseCachingMiddleware - кешування GET запитів на 5 хвилин
- AuthenticationMiddleware - перевірка авторизації

ServerEditorViewModel – повна інтеграція з сервером, підключення/відключення від сервера, завантаження зображень, створення колажів, синхронізація даних.

ServerEditorPage.xaml – інтерфейс з індикатором підключення, панеллю операцій, списком зображень з сервера, списком колажів, логами активності.

Були створені класи для передачі даних між клієнтом і сервером:

- ImageDto - інформація про зображення
- CollageDto - інформація про колаж
- ImageLayerDto - шар зображення в колажі
- EffectDto - застосовані ефекти

```

[ApiController]
[Route("api/[controller]")]
3 references
public class CollagesController : ControllerBase
{
    private readonly ICollageService _collageService;
    private readonly ILogger<CollagesController> _logger;

    0 references
    public CollagesController(ICollageService collageService, ILogger<CollagesController> logger)
    {
        _collageService = collageService;
        _logger = logger;
    }

    [HttpPost]
    0 references
    public async Task<ActionResult<ApiResponse<CollageDto>>> CreateCollage([FromBody] CreateCollageRequest request)
    {
        try
        {
            _logger.LogInformation("Creating collage: {Title}", request.Title);

            var collage = new Collage
            {
                Title = request.Title,
                Width = request.Width,
                Height = request.Height
            };

            var savedCollage = await _collageService.AddAsync(collage);

            var dto = new CollageDto
            {
                Id = savedCollage.Id,
                Title = savedCollage.Title,
                Width = savedCollage.Width,
                Height = savedCollage.Height
            };

            return Ok(new ApiResponse<CollageDto>
            {
                Success = true,
                Message = "Collage created successfully",
                Data = dto
            });
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error creating collage");
        }
    }
}

```

Рисунок 2 – клас CollagesController.cs

CollagesController – контролер для управління колажами:

- POST /api/collages - створення колажу
- GET /api/collages - отримання списку колажів
- GET /api/collages/{id} - отримання конкретного колажу
- POST /api/collages/layer - додавання шару
- DELETE /api/collages/{id} - видалення колажу

```

[ApiController]
[Route("api/[controller]")]
3 references
public class ImagesController : ControllerBase
{
    private readonly IImageService _imageService;
    private readonly ILogger<ImagesController> _logger;

    0 references
    public ImagesController(IImageService imageService, ILogger<ImagesController> logger)
    {
        _imageService = imageService;
        _logger = logger;
    }

    [HttpPost("upload")]
    0 references
    public async Task<ActionResult<ApiResponse<ImageDto>>> UploadImage([FromBody] UploadImageRequest request)
    {
        try
        {
            _logger.LogInformation("Uploading image: {FileName}", request.FileName);

            byte[] imageBytes = Convert.FromBase64String(request.Base64Data);
            string uploadPath = Path.Combine("uploads", request.FileName);
            Directory.CreateDirectory("uploads");
            await System.IO.File.WriteAllBytesAsync(uploadPath, imageBytes);

            var image = new Image
            {
                FilePath = uploadPath,
                Format = Path.GetExtension(request.FileName).TrimStart('.'),
                Width = 0,
                Height = 0,
                CreatedAt = DateTime.UtcNow
            };

            var savedImage = await _imageService.AddAsync(image);

            var dto = new ImageDto
            {
                Id = savedImage.Id,
                FilePath = savedImage.FilePath,
                Format = savedImage.Format,
                Width = savedImage.Width,
                Height = savedImage.Height.
            };
        }
    }
}

```

Рисунок 3 – клас ImagesController.cs

ImagesController – контролер для управління зображеннями:

- POST /api/images/upload - завантаження зображення
- GET /api/images - отримання списку зображень
- GET /api/images/{id} - отримання конкретного зображення
- DELETE /api/images/{id} - видалення зображення

Контролери використовують існуючі сервіси через Dependency Injection:

- IImageService - робота з зображеннями
- ICollageService - робота з колажами
- IEfectService - застосування ефектів

```

4 references
public class ErrorHandlingMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger<ErrorHandlingMiddleware> _logger;

    0 references
    public ErrorHandlingMiddleware(RequestDelegate next, ILogger<ErrorHandlingMiddleware> logger)
    {
        _next = next;
        _logger = logger;
    }

    0 references
    public async Task InvokeAsync(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "An unhandled exception occurred");
            await HandleExceptionAsync(context, ex);
        }
    }
}

1 reference
private static async Task HandleExceptionAsync(HttpContext context, Exception exception)
{
    context.Response.ContentType = "application/json";
    context.Response.StatusCode = StatusCodes.Status500InternalServerError;

    var response = new
    {
        Success = false,
        Message = "An internal server error occurred",
        Detail = exception.Message,
        Timestamp = DateTime.UtcNow
    };

    var jsonResponse = JsonSerializer.Serialize(response);
    await context.Response.WriteAsync(jsonResponse);
}

```

Рисунок 4 – клас ErrorHandlingMiddleware.cs

Створено п'ять middleware компонентів, які обробляють запити в певному порядку:

ErrorHandlingMiddleware – перехоплює всі необроблені винятки і повертає стандартизовану відповідь про помилку. Логує всі помилки для подальшого аналізу.

RequestLoggingMiddleware – логує кожен запит з наступною інформацією: унікальний ID запиту (GUID), HTTP метод (GET, POST, DELETE), шлях запиту, час початку і завершення, час виконання в мілісекундах, HTTP статус код відповіді.

CorsMiddleware – налаштовує Cross-Origin Resource Sharing для дозволу запитів з клієнтського додатку:

- Дозволяє запити з будь-яких джерел
- Підтримує методи GET, POST, PUT, DELETE, OPTIONS
- Дозволяє заголовки Content-Type та Authorization

ResponseCachingMiddleware - кешує відповіді GET запитів на 5 хвилин для оптимізації продуктивності:

- Зберігає відповіді в пам'яті
- Автоматично видаляє прострочені записи
- Пропускає POST/PUT/DELETE запити

```
public class Program
{
    0 references
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        // Додаємо сервіси
        builder.Services.AddControllers();
        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen();

        // Додаємо DbContext
        builder.Services.AddDbContext<LuminaContext>(options =>
            options.UseSqlite("Data Source=lumina.db"));

        // Реєструємо репозиторії та сервіси
        builder.Services.AddScoped(typeof(IRepository<>), typeof(Repository<>));
        builder.Services.AddScoped<IImageService, ImageService>();
        builder.Services.AddScoped<ICollageService, CollageService>();
        builder.Services.AddScoped<IEffectService, EffectService>();

        // Додаємо логування
        builder.Logging.AddConsole();
        builder.Logging.AddDebug();

        var app = builder.Build();

        // Налаштовуємо Middleware pipeline
        if (app.Environment.IsDevelopment())
        {
            app.UseSwagger();
            app.UseSwaggerUI();
        }

        // Власні Middleware (порядок важливий!)
        app.UseMiddleware<ErrorHandlerMiddleware>();
        app.UseMiddleware<RequestLocalizationMiddleware>();
    }
}
```

Рисунок 5 – клас Program.cs

```

3 references
public class ApiClientService
{
    private readonly HttpClient _httpClient;
    private readonly string _baseUrl;

    1 reference
    public ApiClientService(string baseUrl = "https://localhost:7001")
    {
        _baseUrl = baseUrl;
        _httpClient = new HttpClient
        {
            BaseAddress = new Uri(_baseUrl),
            Timeout = TimeSpan.FromSeconds(30)
        };
    }

    3 references
    public IImageApiClient Images => new ImageApiClient(_httpClient);
    3 references
    public ICollageApiClient Collages => new CollageApiClient(_httpClient);
}

```

Рисунок 6 – клас ApiClientService.cs

ApiClientService - головний клас для комунікації.

```

2 references
public class CollageApiClient : ICollageApiClient
{
    private readonly HttpClient _httpClient;

    1 reference
    public CollageApiClient(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    2 references
    public async Task<ApiResponse<CollageDto>> CreateCollageAsync(string title, int width, int height)
    {
        try
        {
            var request = new CreateCollageRequest
            {
                Title = title,
                Width = width,
                Height = height
            };

            var response = await _httpClient.PostAsJsonAsync("/api/collages", request);
            response.EnsureSuccessStatusCode();

            return await response.Content.ReadFromJsonAsync<ApiResponse<CollageDto>>()
                ?? new ApiResponse<CollageDto>{ Success = false, Message = "Failed to parse response" };
        }
        catch (Exception ex)
        {
            return new ApiResponse<CollageDto>
            {
                Success = false,
                Message = $"Error creating collage: {ex.Message}"
            };
        }
    }
}

```

Рисунок 7 – клас CollageApiClient.cs

CollageApiClient - реалізує методи для роботи з колажами:

- CreateCollageAsync() - створює новий колаж
- GetAllCollagesAsync() - отримує список колажів
- AddLayerAsync() - додає шар до колажу

```

public class ImageApiClient : IImageApiClient
{
    private readonly HttpClient _httpClient;

    public ImageApiClient(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

2 references
public async Task<ApiResponse<ImageDto>> UploadImageAsync(string filePath)
{
    try
    {
        // Читаемо файл і конвертуємо в Base64
        byte[] imageBytes = await File.ReadAllBytesAsync(filePath);
        string base64Data = Convert.ToBase64String(imageBytes);

        var request = new UploadImageRequest
        {
            FileName = Path.GetFileName(filePath),
            Base64Data = base64Data
        };

        var response = await _httpClient.PostAsJsonAsync("/api/images/upload", request);
        response.EnsureSuccessStatusCode();

        return await response.Content.ReadFromJsonAsync<ApiResponse<ImageDto>>()
            ?? new ApiResponse<ImageDto> { Success = false, Message = "Failed to parse response" };
    }
    catch (Exception ex)
    {
        return new ApiResponse<ImageDto>
        {
            Success = false,
            Message = $"Error uploading image: {ex.Message}"
        };
    }
}

3 references
public async Task<ApiResponse<List<ImageDto>>> GetAllImagesAsync()
{
    try
    {
        var response = await _httpClient.GetAsync("/api/images");
        response.EnsureSuccessStatusCode();
    }
}

```

Рисунок 8 – клас ImageApiClient.cs

ImageApiClient - реалізує методи для роботи з зображеннями:

- UploadImageAsync() - читає файл, конвертує в Base64 і відправляє
- GetAllImagesAsync() - отримує список усіх зображень
- GetImageByIdAsync() - завантажує конкретне зображення
- DeleteImageAsync() - видаляє зображення

```

1 reference
public class ServerEditorViewModel : INotifyPropertyChanged
{
    private readonly ApiClientService _apiClient;

    private string _logText = "";
1 reference
    public string LogText
    {
        get => _logText;
        set { _logText = value; OnPropertyChanged(); }
    }

    private bool _isConnected = false;
3 references
    public bool IsConnected
    {
        get => _isConnected;
        set { _isConnected = value; OnPropertyChanged(); }
    }

    private string _connectionStatus = "Disconnected";
3 references
    public string ConnectionStatus
    {
        get => _connectionStatus;
        set { _connectionStatus = value; OnPropertyChanged(); }
    }

    private int? _currentCollageId = null;
7 references
    public int? CurrentCollageId
    {
        get => _currentCollageId;
        set { _currentCollageId = value; OnPropertyChanged(); }
    }

    3 references
    public ObservableCollection<ImageDto> ServerImages { get; set; } = new();
    3 references
    public ObservableCollection<CollageDto> ServerCollages { get; set; } = new();

    1 reference
    public ICommand ConnectToServerCommand { get; }
    1 reference
    public ICommand UploadImageCommand { get; }

    public ICommand LoadImagesCommand { get; }
    1 reference
    public ICommand CreateCollageCommand { get; }

    public ICommand LoadCollagesCommand { get; }
}

```

Рисунок 9 – клас ServerEditorViewModel.cs

ServerEditorViewModel містить:

- Властивості для відображення стану з'єднання
- Колекції ServerImages і ServerCollages для відображення даних
- Команди для всіх операцій (Connect, Upload, Create тощо)
- Логіку обробки відповідей та помилок

```

psa@DESKTOP-LTIHJ5F MINGW4 ~/source/repos/Lumina/Lumina/Lumina.Server (main)
$ dotnet run
Using launch settings from C:\Users\psa\source\repos\Lumina\Lumina\Lumina.Server
\Properties\launchSettings.json...
Building...
C:\Users\psa\source\repos\Lumina\Lumina\Core\Patterns\EditorContext.cs(9,
16): warning CS8618: Non-nullable field '_currentState' must contain a non-null
C:\Users\psa\source\repos\Lumina\Lumina\Server\Middleware\CorsMiddleware.cs(15,13): warning ASP0019: Use IHeaderDictionary.Append or the indexer to append o
row an ArgumentException when attempting to add a duplicate key. (https://aka.ms/aspnet/analyzers)
C:\Users\psa\source\repos\Lumina\Lumina\Server\Middleware\CorsMiddleware.cs(16,13): warning ASP0019: Use IHeaderDictionary.Append or the indexer to append o
row an ArgumentException when attempting to add a duplicate key. (https://aka.ms/aspnet/analyzers)
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5155
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\psa\source\repos\Luminari\Lumina\Lumina.Server
info: Lumina.Server.Middleware.RequestLoggingMiddleware[0]
      Request [022bb850-a-ca-41e7-afdf-a/f5849cc68]: GET / started at 12/09/2025 07:50:39
warn: Microsoft.AspNetCore.HttpsPolicy.HttpsRedirectionMiddleware[3]
      Failed to determine the https port for redirect.
info: Lumina.Server.Middleware.RequestLoggingMiddleware[0]

```

Рисунок 10 – Запуск сервера

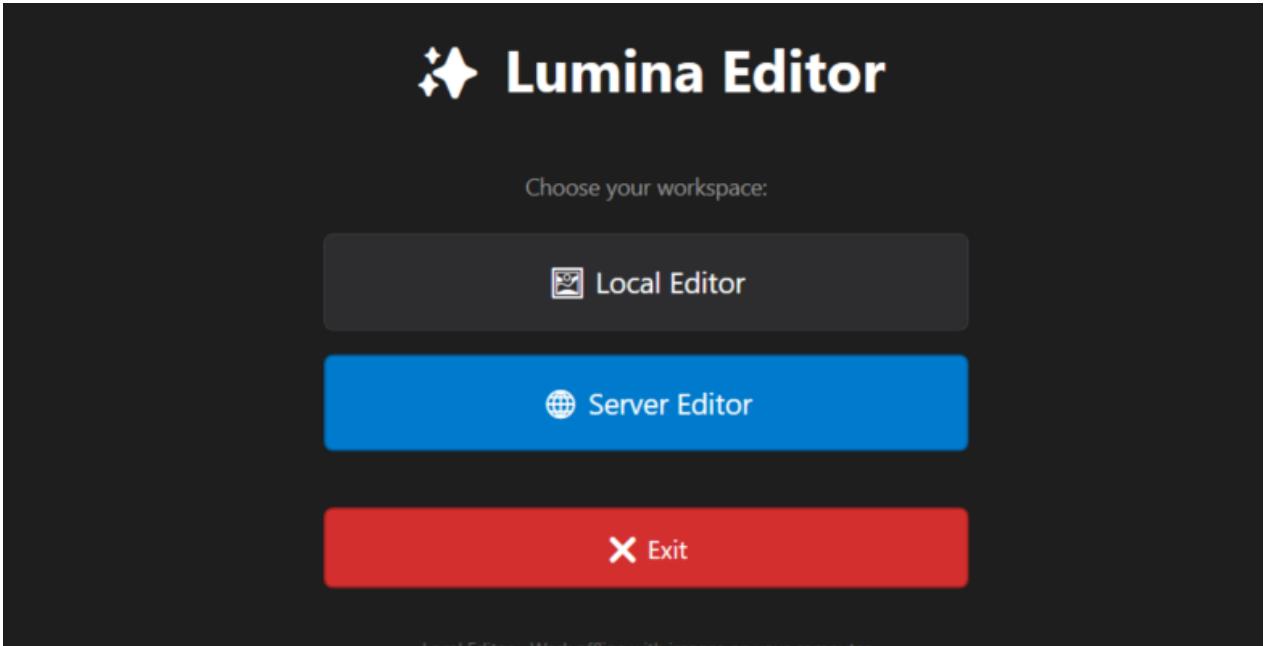


Рисунок 11 – Головна сторінка

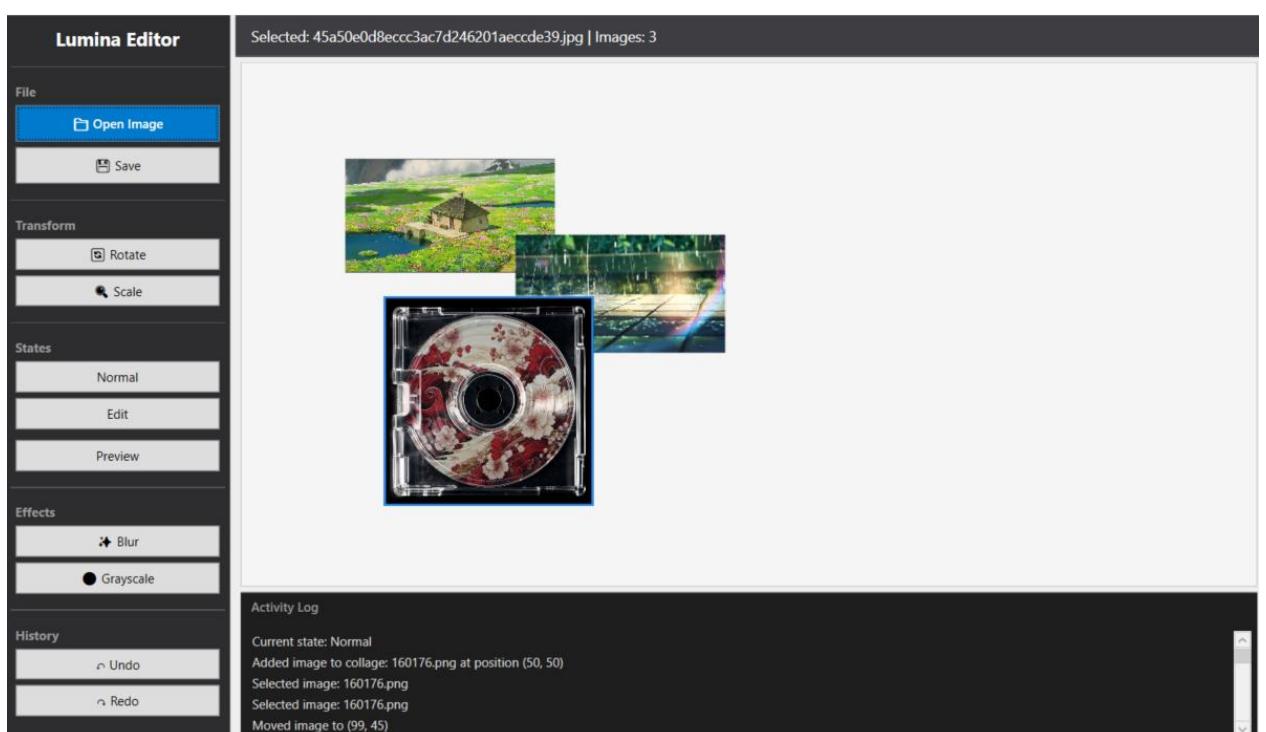


Рисунок 12 – Локальна сторінка редактора

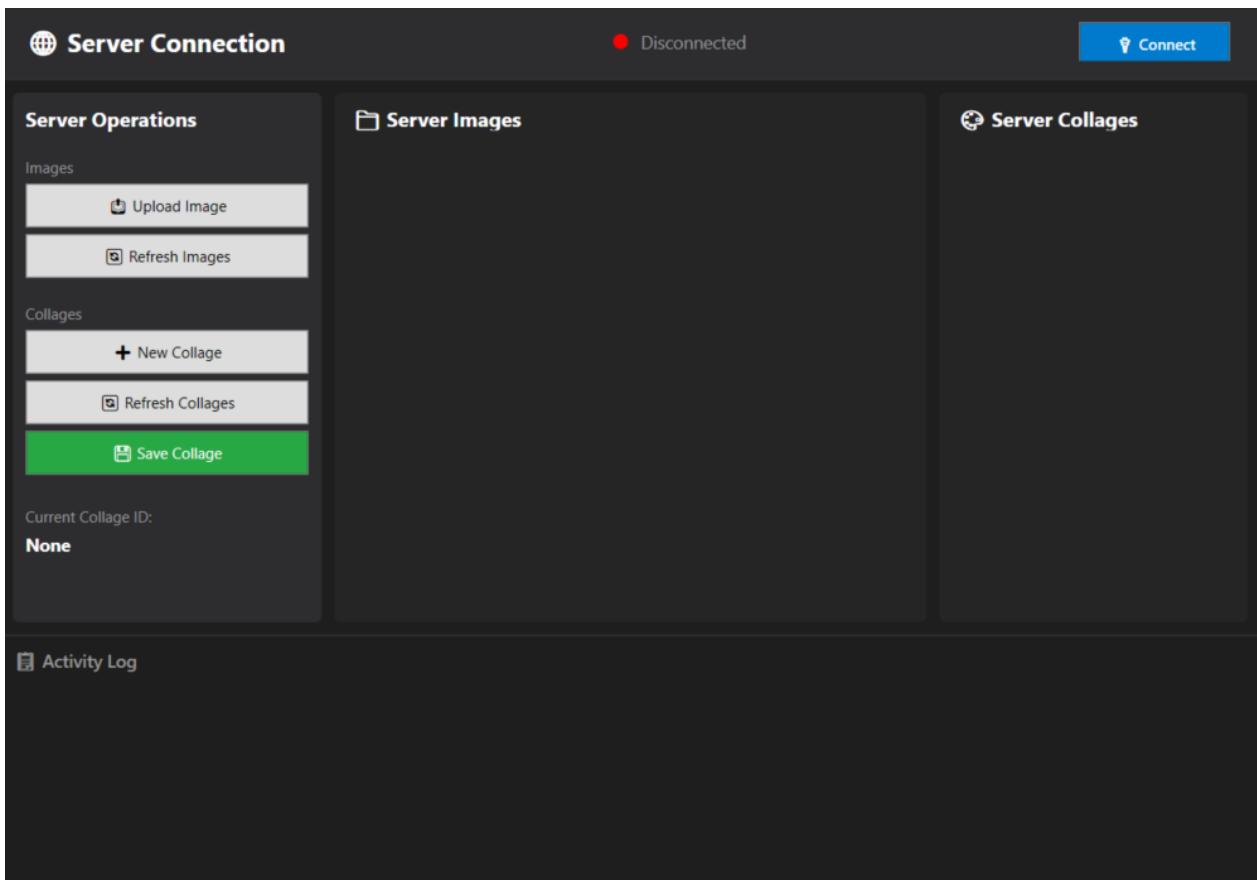


Рисунок 13 – Серверна сторінка редактора

The screenshot shows the Swagger UI interface for the Lumina.Server API. At the top, it says "Swagger" and "Supported by SMARTBEAR". A dropdown menu says "Select a definition" with "Lumina.Server v1" selected. Below that, the title "Lumina.Server" is shown with "1.0 OAS 3.0" and the URL "http://localhost:5155/swagger/v1/swagger.json". The interface is organized into sections: "Collages" and "Images". Under "Collages", there are four API endpoints: a green "POST /api/Collages", a blue "GET /api/Collages", a blue "GET /api/Collages/{id}", and a red "DELETE /api/collages/{id}" which is expanded to show its details. Under "Images", there are four API endpoints: a green "POST /api/Images/upload", a blue "GET /api/Images", a blue "GET /api/Images/{id}", and a red "DELETE /api/Images/{id}" which is also expanded. Each endpoint row has a collapse arrow at the end.

Рисунок 14 –Swagger UI для тестування API ендпоїнтів

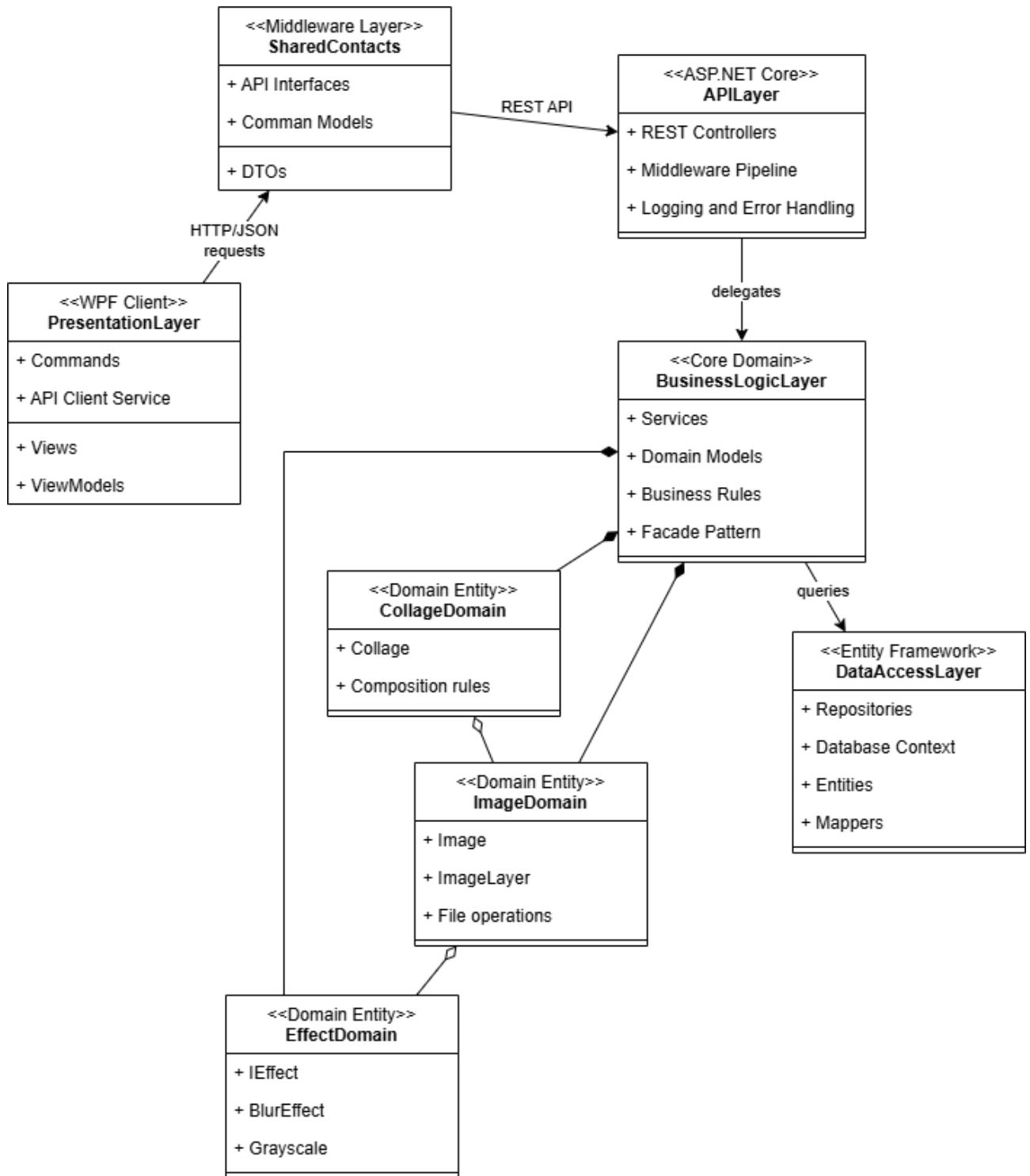


Рисунок 15 – Діаграма класів, яка представляє спроектовану архітектуру

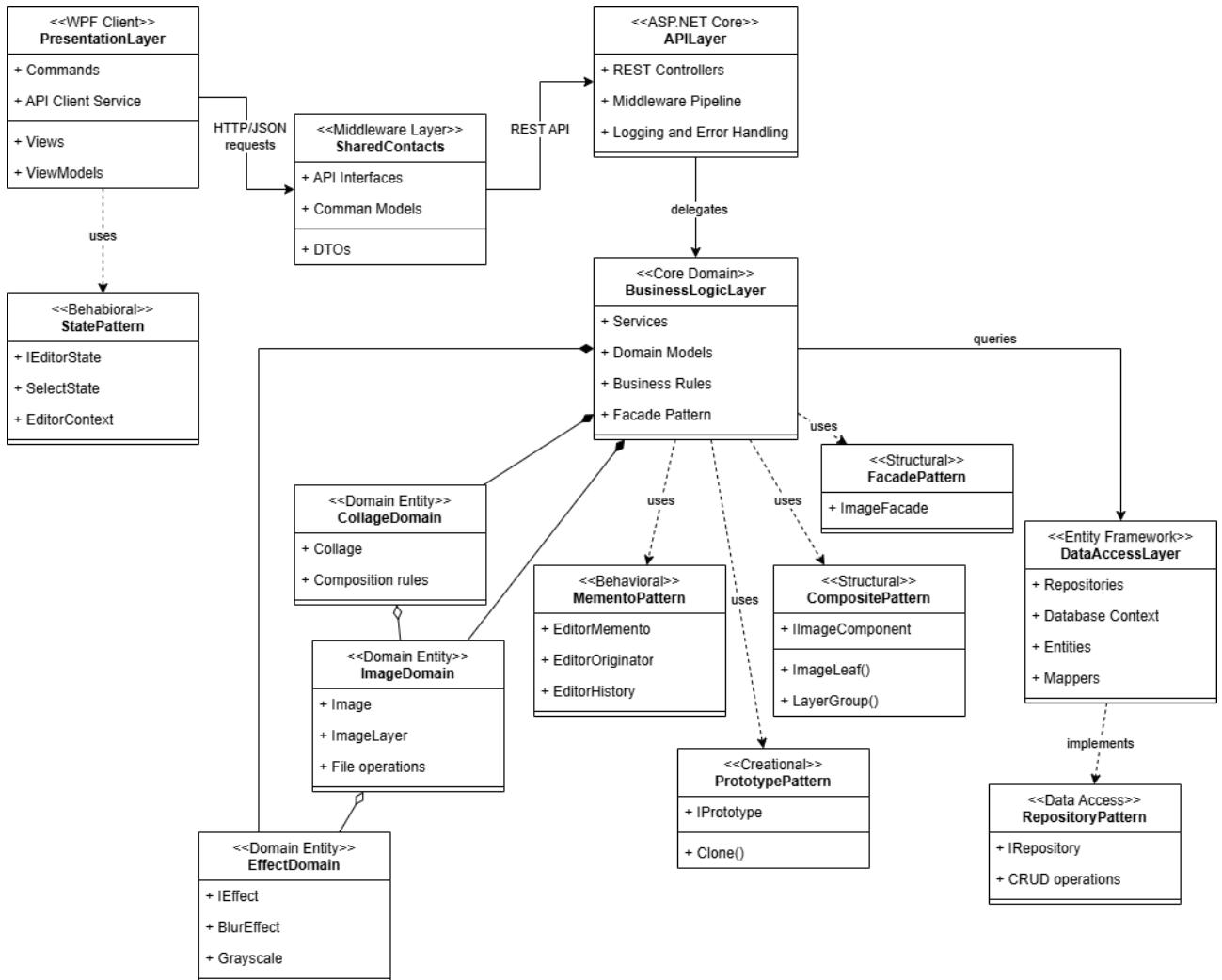


Рисунок 16 – Діаграма класів, яка представляє спроектовану архітектуру (з патернами)

Клієнт-серверна архітектура забезпечує чіткий розподіл обов'язків між частинами системи: клієнт відповідає за інтерфейс та взаємодію з користувачем, сервер – за бізнес-логіку та обробку даних, а middleware виступає посередником, що забезпечує коректний обмін інформацією. Такий підхід дає можливість масштабувати серверну частину, підключати необмежену кількість клієнтів і незалежно розробляти різні інтерфейси – настільний, веб або мобільний.

Middleware у системі виконує функції обробки помилок, логування, CORS-налаштувань, кешування та автентифікації. Правильний порядок розміщення цих компонентів гарантує стабільну й передбачувану роботу серверної частини.

Спільна бібліотека моделей та контрактів забезпечує сумісність між клієнтом і сервером і запобігає конфліктам при серіалізації.

Реалізована система має практичне застосування в сценаріях, де потрібна робота зображень у розподіленому середовищі: хмарне зберігання проектів, синхронізація між пристроями, спільне редагування або створення багатоплатформових клієнтів. Архітектура легко розширюється новими функціями, контролерами або middleware.

Висновки

Клієнт-серверний підхід значно спрощує розробку великих систем, дозволяючи розділити функції між окремими частинами та забезпечити незалежність їх розвитку.

Middleware відіграє ключову роль у системі: відповідає за стабільність, безпеку й продуктивність. Його впорядковане використання робить архітектуру передбачуваною та гнучкою.

Спільні моделі та контракти дозволили забезпечити повну сумісність клієнта та сервера й спростили роботу з даними.

Реалізована система демонструє практичну придатність до використання в реальних умовах: від хмарного редагування зображень до роботи кількох клієнтів одночасно.

Завдяки продуманій архітектурі система є масштабованою, розширюваною та безпечною, що дозволяє надалі доповнювати її новими можливостями без радикальних змін у кодовій базі.

Контрольні питання

1. Що таке клієнт-серверна архітектура?

Клієнт-серверна архітектура – це модель розподіленої системи, де чітко розділені ролі між клієнтами та серверами. Клієнти відповідають за представлення даних та взаємодію з користувачем, а сервери забезпечують зберігання, обробку даних та виконання бізнес-логіки. Ця архітектура може реалізовуватися у вигляді тонких клієнтів (більшість обробки на сервері), товстих клієнтів (значна частина логіки на клієнті) або проміжних рішень на кшталт SPA-додатків.

2. Розкажіть про сервіс-орієнтовану архітектуру.

Сервіс-орієнтована архітектура (SOA) – це модульний підхід до розробки програмного забезпечення, заснований на використанні розподілених, слабко пов'язаних сервісів зі стандартизованими інтерфейсами. Вона виникла як альтернатива монолітним системам і зазвичай реалізується через веб-сервіси, що взаємодіють по протоколах SOAP або REST. Кожен сервіс інкапсулює певну бізнес-функцію і взаємодіє з іншими сервісами виключно через обмін повідомленнями.

3. Якими принципами керується SOA?

- Слабка зв'язаність сервісів
- Багаторазовість компонентів
- Стандартизовані контракти спілкування
- Автономність сервісів
- Використання стандартних протоколів
- Реєстрація та виявлення сервісів
- Абстрагування від реалізації

4. Як між собою взаємодіють сервіси в SOA?

Сервіси в SOA взаємодіють виключно через стандартизовані інтерфейси за допомогою обміну повідомленнями. Вони не розділяють спільні ресурси

(наприклад, бази даних) і не мають прямих залежностей між собою. Для оркестрації взаємодії часто використовується шина корпоративних сервісів (ESB), яка забезпечує трансформацію повідомлень, маршрутизацію та моніторинг.

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

В SOA існує механізм реєстрації та виявлення сервісів. Сервіси реєструються в спеціальних реєстрах або сховищах, де вони описують свої інтерфейси, контракти та способи доступу. Розробники можуть звертатися до цих реєстрів для пошуку потрібних сервісів, отримувати інформацію про їхні інтерфейси та використовувати стандартизовані протоколи (як SOAP чи REST) для взаємодії з ними.

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

Переваги:

- Централізоване управління та безпека
- Просте розгортання та оновлення (для тонких клієнтів)
- Ефективне використання ресурсів сервера
- Надійність та узгодженість даних

Недоліки:

- Ризик єдиної точки відмови
- Можливі проблеми з масштабуванням Залежність від мережевого з'єднання
- Високе навантаження на сервер при великій кількості клієнтів

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

Переваги:

- Відсутність єдиної точки відмови
- Висока масштабованість
- Ефективне використання ресурсів усіх учасників
- Стійкість до цензури та обмежень

Недоліки:

- Складність забезпечення безпеки
- Проблеми з синхронізацією даних
- Складність пошуку ресурсів у великих мережах
- Юридичні ризики та проблеми з контролем контенту

8. Що таке мікро-сервісна архітектура?

Мікросервісна архітектура – це підхід до розробки програмного забезпечення, при якому додаток складається з набору дрібних, автономних сервісів, кожен з яких виконує конкретну бізнес-функцію. Кожен мікросервіс має чітко визначені межі, може розгортатися незалежно та взаємодіє з іншими сервісами через легкі протоколи. Ця архітектура є еволюційним розвитком SOA з акцентом на дрібніші компоненти та повну автономність.

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

- HTTP/HTTPS з REST API
- gRPC для високопродуктивних зв'язків
- AMQP (RabbitMQ) для асинхронної взаємодії
- WebSockets для реального часу
- Apache Kafka для потокової обробки даних
- GraphQL для гнучких запитів даних

10. Чи можна назвати підхід сервіс-орієнтованою архітектурою, коли ми в проекті між шаром веб-контролерів та шаром доступу до даних реалізуємо шар бізнес-логіки у вигляді сервісів?

Ні, такий підхід не можна вважати справжньою сервіс-орієнтованою архітектурою. Це швидше демонструє шаблон "сервісного шару" у монолітній архітектурі. Справжня SOA передбачає:

- Розподілені, незалежно розгорнуті сервіси
- Слабку зв'язаність між компонентами
- Стандартизовани інтерфейси спілкування

- Можливість повторного використання сервісів у різних додатках
- Механізми виявлення та реєстрації сервісів

Просто виділення бізнес-логіки в окремі класи-сервіси всередині одного додатку є хорошою практикою, але не становить повноцінної SOA.