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: «Взаємодія компонентів системи.»

«Менеджер завдань»

Виконав:
студент групи - ІА-32
Воробйов Кирило Андрійович

Перевірив:
Мягкий Михайло Юрійович

Київ 2025

Тема: Взаємодія компонентів системи.

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

Тема проєкту: Менеджер завдань (To Do Manager)

Патерни: Strategy, Command, Observer, Template Method, Composite, Client-Server

Опис: Додаток повинен мати можливість створювати, редагувати та видаляти завдання, забезпечувати гнучку фільтрацію та сортування (Strategy), та обробляти дії користувача (Command). Система має підтримувати ієрархічну структуру (Composite), забезпечувати автоматичне оновлення візуальних компонентів при зміні даних (Observer), а також надавати функціонал генерації звітів та експорту даних, використовуючи послідовний алгоритм (Template Method). Зберігання та синхронізація завдань реалізується через клієнт-серверний зв'язок (Client-Server).

Зміст

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.

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.

Основними сферами де peer-to-peer архітектура знайшла широке застосування є файлообмінники (BitTorrent), криптовалюти та інші блокчейнтехнології, 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 [11].

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 [11]. 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 Макларті 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

Завдання:

- Ознайомитись з короткими теоретичними відомостями.
- Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
- Реалізувати функціонал для роботи в розподіленому оточенні відповідно до обраної теми.
- Реалізувати взаємодію розподілених частин:

  Для клієнт-серверних варіантів: реалізація клієнтської і серверної частини додатків, а також загальної частини (middleware); зв'язок клієнтської і серверної частин за допомогою WCF, TcpClient, .NETRemoting на розсуд виконавця.

Діаграма класів патерну Client-Server

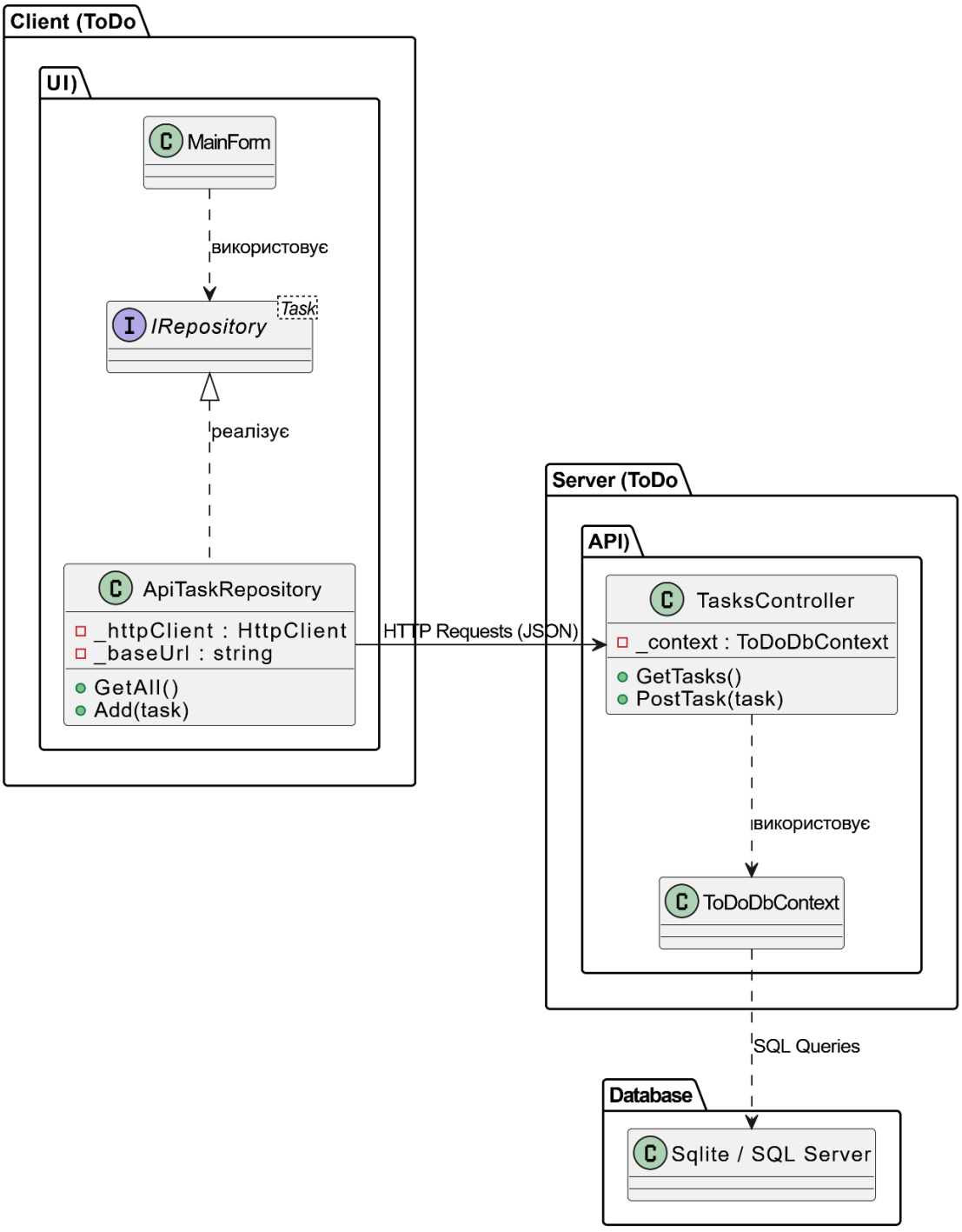


Рис. 1. Діаграма класів патерну Client-Server

Діаграма демонструє розподілену архітектуру системи, яка складається з двох незалежних вузлів: Клієнта (Desktop Application) та Сервера (Web API), що взаємодіють через мережу.

1 **Рівень Клієнта (ToDo.UI):**

- **MainForm:** Головне вікно програми. Воно не знає про існування мережі чи бази даних. Воно працює виключно з абстракцією IRepository.
- **ApiTaskRepository:** Ключовий компонент клієнтської частини. Він реалізує інтерфейс IRepository, але замість прямого доступу до БД, він формує **HTTP-запити** (GET, POST, PUT, DELETE) та відправляє їх на сервер. Він виступає як адаптер між UI та віддаленим API .

2 **Інтерфейс (IRepository<T>):**

- **Забезпечує** слабку зв'язаність. Завдяки йому ми змогли замінити локальну БД на віддалений сервер, не змінюючи жодного рядка коду у візуальних формах (MainForm).

3 **Рівень Сервера (ToDo.API):**

- **TasksController:** Вхідна точка на сервері. Цей клас приймає HTTP-запити від клієнта, обробляє їх та викликає відповідні методи роботи з даними. Він діє як фасад для серверної логіки.
- **ToDoDbContext:** Знаходиться на стороні сервера і єдиний має прямий доступ до бази даних.

4 **Взаємодія:**

- Зв'язок здійснюється через протокол **HTTP** за допомогою обміну даними у форматі **JSON**. Клієнт серіалізує об'єкти в JSON, а сервер десеріалізує їх назад у C# об'єкти для збереження в БД.

Код програми

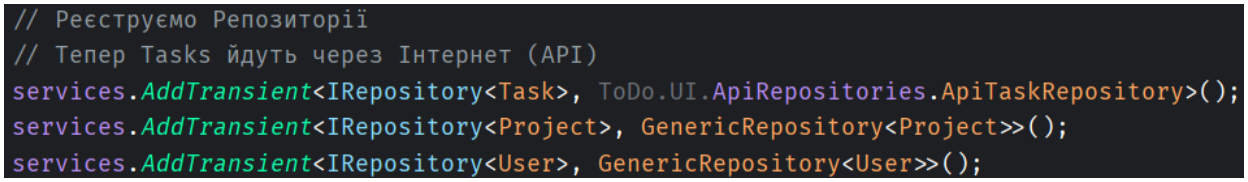
ApiTaskRepository.cs

```
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Net.Http.Json;
using System.Threading.Tasks;
using ToDo.Models;
using ToDo.Repositories;
using Task = ToDo.Models.Task;

namespace ToDo.UI.ApiRepositories
{
    public class ApiTaskRepository : IRepository<Task>
    {
        private readonly HttpClient _httpClient;
        private readonly string _baseUrl = "http://localhost:5260/api/tasks";

        public ApiTaskRepository()
        {
            _httpClient = new HttpClient();
        }

        public IEnumerable<Task> GetAll()
        {
            try
            {
                // Використовуємо синхронне очікування (.Result) для сумісності з інтерфейсом
                return _httpClient.GetFromJsonAsync<List<Task>>(_baseUrl).Result ?? new List<Task>();
            }
            catch (Exception)
            {
                // Якщо сервер вимкнений, повертаємо пустий список, щоб програма не впала
                return new List<Task>();
```

```
        }
    }

    public Task? GetById(int id)
    {
        try
        {
            return
_httpClient.GetFromJsonAsync<Task>($"{_baseUrl}/{id}").Result;
        }
        catch
        {
            return null;
        }
    }

    public void Add(Task entity)
    {
        var response = _httpClient.PostAsJsonAsync(_baseUrl, entity).Result;
        response.EnsureSuccessStatusCode();
    }

    public void Update(Task entity)
    {
        var response =
_httpClient.PutAsJsonAsync($"{_baseUrl}/{entity.TaskId}", entity).Result;
        response.EnsureSuccessStatusCode();
    }

    public void Delete(int id)
    {
        var response = _httpClient.DeleteAsync($"{_baseUrl}/{id}").Result;
        response.EnsureSuccessStatusCode();
    }
  }
}
```

Program.cs

```
// Реєструємо Репозиторії
// Тепер Tasks йдуть через Інтернет (API)
services.AddTransient<IRepository<Task>,
ToDo.UI.ApiRepositories.ApiTaskRepository>();
services.AddTransient<IRepository<Project>, GenericRepository<Project>>();
services.AddTransient<IRepository<User>, GenericRepository<User>>();
```



TasksController.cs

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using ToDo.Data;
using ToDo.Models;
using Task = ToDo.Models.Task;

namespace ToDo.API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class TasksController : ControllerBase
    {
        private readonly ToDoDbContext _context;

        public TasksController(ToDoDbContext context)
        {
            _context = context;
        }

        // GET: api/tasks (Отримати всі завдання)
        [HttpGet]
        public async Task<ActionResult<IEnumerable<Task>>> GetTasks()
        {
            return await _context.Tasks.ToListAsync();
        }
```

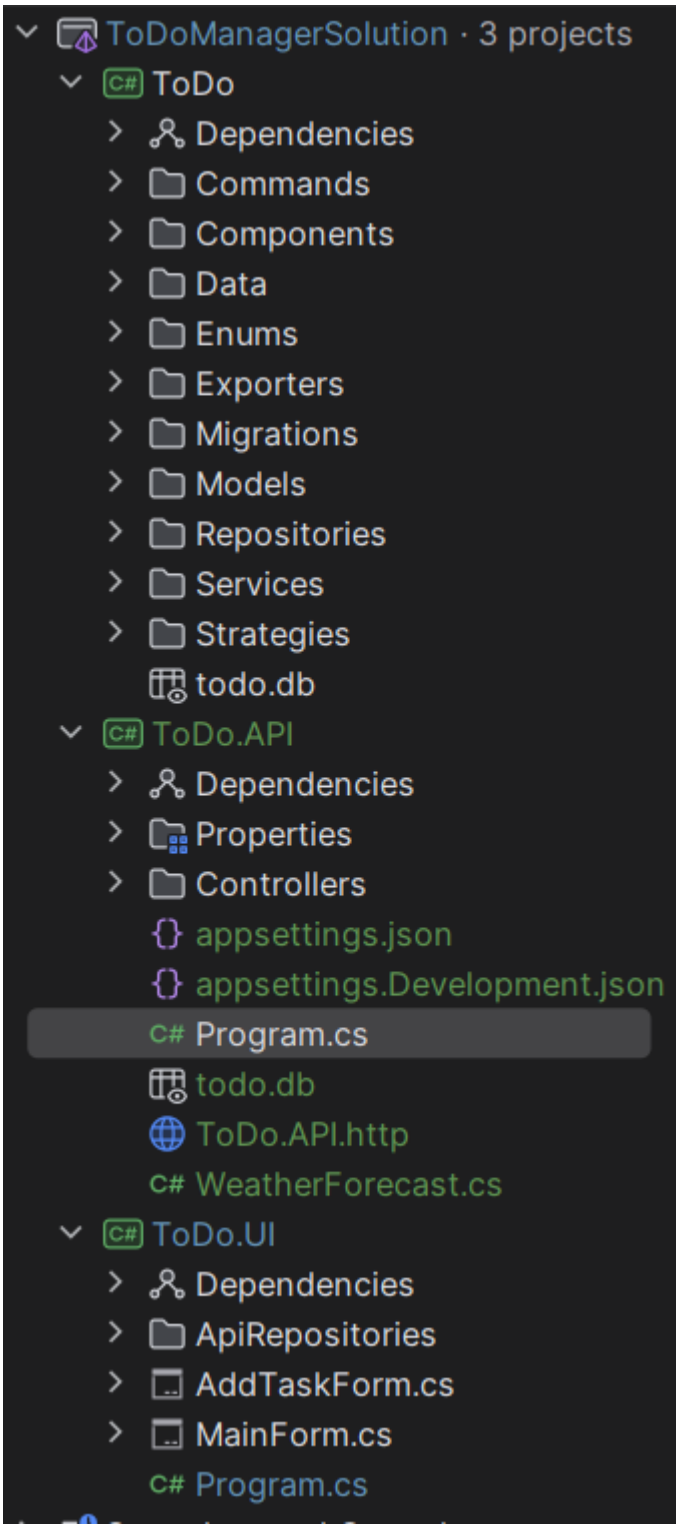
```
    // GET: api/tasks/5 (Отримати одне по ID)
    [HttpGet("{id}")]
    public async Task<ActionResult<Task>> GetTask(int id)
    {
        var task = await _context.Tasks.FindAsync(id);

        if (task == null)
        {
            return NotFound();
        }

        return task;
    }

    // POST: api/tasks (Створити нове)
    [HttpPost]
    public async Task<ActionResult<Task>> PostTask(Task task)
    {
        _context.Tasks.Add(task);
        await _context.SaveChangesAsync();

        return CreatedAtAction(nameof(GetTask), new { id = task.TaskId },
task);
    }

    // PUT: api/tasks/5 (Оновити)
    [HttpPut("{id}")]
    public async Task<IActionResult> PutTask(int id, Task task)
    {
        if (id != task.TaskId)
        {
            return BadRequest();
        }

        _context.Entry(task).State = EntityState.Modified;

        try
        {
```

```
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!_context.Tasks.Any(e => e.TaskId == id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }

        return NoContent();
    }

    // DELETE: api/tasks/5 (Видалити)
    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteTask(int id)
    {
        var task = await _context.Tasks.FindAsync(id);
        if (task == null)
        {
            return NotFound();
        }

        _context.Tasks.Remove(task);
        await _context.SaveChangesAsync();

        return NoContent();
    }
  }
}
```

Program.cs (Сервер)

```
using ToDo.Data;
using Microsoft.EntityFrameworkCore;
using ToDo.Models;

var builder = WebApplication.CreateBuilder(args);

// Реєстрація сервісів
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

// Підключення БД
builder.Services.AddDbContext<ToDoDbContext>();

var app = builder.Build();

// Блок ініціалізації бази даних на сервері
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    try
    {
        var context = services.GetRequiredService<ToDoDbContext>();

        context.Database.Migrate();

        // Створюємо базові дані (User/Project), інакше при створенні Task
виникне помилка ключа
        if (!context.Users.Any())
        {
            var defaultUser = new User
            {
                Username = "server_user",
                PasswordHash = "hashedpassword"
            };
            context.Users.Add(defaultUser);
            context.SaveChanges();
```

```
            // Створюємо проєкт для цього юзера
            context.Projects.Add(new Project
            {
                Title = "Server Project",
                CreationDate = DateTime.Now,
                UserId = defaultUser.UserId
            });
            context.SaveChanges();
        }
    }
    catch (Exception ex)
    {
        var logger = services.GetRequiredService<ILogger<Program>>();
        logger.LogError(ex, "An error occurred while seeding the database.");
    }
}

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseAuthorization();

app.MapControllers();

app.Run();
```


ToDoManagerSolution · 3 projects
ToDo
Dependencies
Commands
Components
Data
Enums
Exporters
Migrations
Models
Repositories
Services
Strategies
todo.db
ToDo.API
Dependencies
Properties
Controllers
appsettings.json
appsettings.Development.json
Program.cs
todo.db
ToDo.API.http
WeatherForecast.cs
ToDo.UI
Dependencies
ApiRepositories
AddTaskForm.cs
MainForm.cs
Program.cs

Висновки

Висновки: під час виконання лабораторної роботи ми трансформували монолітний додаток у клієнт-серверну архітектуру. Було розроблено REST API сервер на базі ASP.NET Core, який забезпечує централізований доступ до даних. Клієнтську частину було модифіковано для взаємодії із сервером через протокол HTTP за допомогою патерна Repository (клас ApiTaskRepository), що дозволило зберегти існуючий інтерфейс користувача без змін

Питання до лабораторної роботи

1. **Що таке клієнт-серверна архітектура?**

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

2. **Розкажіть про сервіс-орієнтовану архітектуру.**

Сервіс-орієнтована архітектура (SOA) – це модульний підхід до розробки програмного забезпечення, заснований на використанні **розподілених, слабо пов’язаних сервісів** (служб), оснащених стандартизованими інтерфейсами для взаємодії за стандартизованими протоколами. Вона історично з'явилася як альтернатива монолітній архітектурі.

3. **Якими принципами керується SOA?**

SOA керується принципами використання **розподілених сервісів**, які є **слабо пов'язаними** (Loose coupling) та оснащені **стандартизованими інтерфейсами** для взаємодії.

4. **Як між собою взаємодіють сервіси в SOA?**

Сервіси взаємодіють між собою тільки за рахунок **обміну повідомленнями**, без створення спеціальних інтеграцій для доступу до

однієї інформації (наприклад, до однієї бази даних). Зазвичай вони взаємодіють по HTTP з використанням **SOAP** або **REST**.

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

Згідно з SOA, сервіси **реєструються на спеціальних сервісах** (реєстрах), і будь-яка команда розробників, якій потрібен доступ, може знайти їх там та використовувати.

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

- **Переваги:** Простота розгортання (оновлювати потрібно лише сервери). Якщо використовується "товстий клієнт", це розвантажує сервер і дає можливість працювати без тимчасового доступу до сервера.

- **Недоліки:** У варіанті "тонкого клієнта" майже все навантаження лягає на сервер. SPA-застосунки не працюють, якщо сервер не доступний.

7. **У чому полягають переваги та недоліки одноранговової моделі взаємодії?**

- **Переваги:** Децентралізація (зменшує залежність від одного вузла, підвищує стійкість до збоїв і атак). Розподіл ресурсів (вузли надають свої обчислювальні потужності або дисковий простір).

- **Недоліки:** Проблеми з безпекою, синхронізацією даних та пошуком ресурсів. Складно контролювати дані через децентралізацію. Ефективність пошуку знижується зі збільшенням кількості вузлів.

8. **Що таке мікро-сервісна архітектура?**

Це підхід до створення серверного додатку як **набору малих служб**. Кожен мікросервіс є компонентом із чітко визначеними межами, який можна розгортати незалежно, виконується в своєму процесі і реалізує специфічні можливості в предметній області .

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

Використовуються такі протоколи, як **HTTP/HTTPS**, **WebSockets** чи **AMQP** (для зв'язку на основі повідомлень).

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. SOA базується на використанні **розподілених** сервісів, що взаємодіють за **стандартизованими протоколами** (як-от SOAP/REST по HTTP). Якщо сервіси знаходяться всередині одного монолітного проєкту і викликаються як звичайні класи коду, це не відповідає визначенню розподіленої SOA, яка виникла як альтернатива монолітній архітектурі.