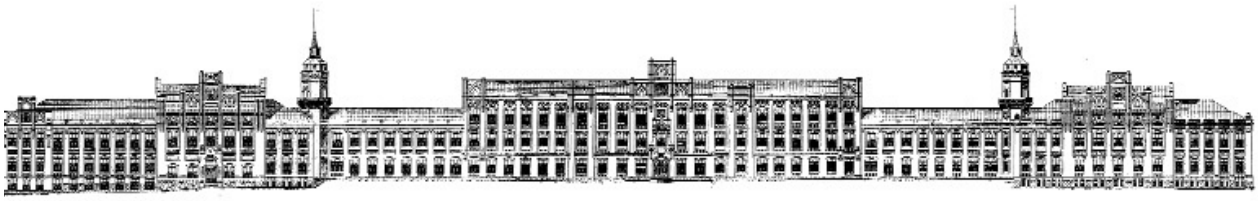

Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Звіт

З дисципліни “Технології розроблення програмного забезпечення”

Варіант: 23

Виконала
студентка групи ІА-23:
Павленко Анастасія

Перевірив:
Мягкий Михайло
Юрійович

Київ 2024

Лабораторна робота №9

Різні види взаємодії додатків: client-server, peer-to-peer, service-oriented architecture

Мета лабораторної роботи: Метою цієї лабораторної роботи є набуття практичних навичок створення програмного забезпечення для управління проєктами, яке працює в розподіленому середовищі, а також розуміння основних підходів до взаємодії клієнт-сервер, peer-to-peer та сервіс-орієнтованої архітектури (SOA). Студенти мають навчитися розробляти клієнтську, серверну та проміжну (middleware) частини додатків, організовувати зв’язок між ними за допомогою технологій WCF, TcpClient або .NET Remoting.

Варіант 23: Project Management software (proxy, chain of responsibility, abstract factory, bridge, flyweight, client-server)

Програмне забезпечення для управління проектами повинно мати наступні функції: супровід завдань/вимог/проектів, списків команд, поточних завдань, планування за методологіями agile/kanban/rup (включаючи дошку завдань, ітерації тощо), мати можливість прикріплювати вкладені файли до завдань та посилатися на конкретні версії програми, зберігати виконувані файли для кожної версії.

Завдання:

1. Реалізувати функціонал для роботи в розподіленому оточенні (логіку роботи).

2. Реалізувати взаємодію розподілених частин.

А) 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 або .net-remoting на розсуд виконавця;

Короткі теоретичні відомості

Клієнт-серверна архітектура

Клієнт – комп'ютер на стороні користувача, який відправляє запит до сервера для надання інформації або виконання певних дій.

Сервер – більш потужний комп'ютер або обладнання, призначене для вирішення певних завдань з виконання програмних кодів, виконання сервісних функцій за запитом клієнтів, надання користувачам доступу до певних ресурсів, зберігання інформації і баз даних.

Модель такої системи полягає в тому, що клієнт відправляє запит на сервер, де він обробляється, і готовий результат відправляється клієнтові. Сервер може обслуговувати кілька клієнтів одночасно. Якщо одночасно приходить більше одного запиту, то вони встановлюються в чергу і виконуються сервером послідовно. Іноді запити можуть мати пріоритети. Запити з більш високими пріоритетами повинні виконуватися раніше.

Функції, які реалізуються на сервері:

- зберігання, доступ, захист і резервне копіювання даних;
- обробка клієнтського запиту;
- відправлення результату (відповіді) клієнту.

Функції, які реалізуються на стороні клієнта:

- надання користувальницького інтерфейсу;
- формулювання запиту до сервера і його відправка;
- отримання результатів запиту і відправка додаткових команд (запитів на додавання, оновлення або видалення даних).

Архітектура клієнт-сервер визначає принципи спілкування між комп'ютерами, а правила і взаємодії визначені в протоколі.

Мережевий протокол – це набір правил, за якими відбувається взаємодія між комп'ютерами в мережі.

Мережеві протоколи:

TCP/IP – набір (стек) протоколів передачі даних. TCP/IP – це позначення всієї мережі, яка працює на основі двох протоколів – TCP і IP.

TCP (Transfer Control Protocol) – протокол, який служить для встановлення надійного з'єднання між двома пристроями, передачі інформації і підтвердження її отримання.

IP (Internet Protocol) – інтернет протокол, який відповідає за правильність доставки повідомлень за певною адресою. При цьому дані розбиваються на пакети, які можуть доставлятися по-різному.

MAC (Media Access Control) – протокол, за допомогою якого відбувається ідентифікація мережевих пристроїв. Всі пристрої, підключені до інтернету, мають свою унікальну MAC адресу.

ICMP (Internet control message protocol) – протокол, який відповідає за обмін інформацією, але не використовується для передачі даних.

HTTP (Hyper Text Transfer Protocol) – протокол передачі гіпертексту, на основі якого працюють всі сайти. Він запитує необхідні дані у віддаленій системи (веб-сторінки, файли).

FTP (File Transfer Protocol) – протокол передачі файлів зі спеціального файлового сервера на комп'ютер користувача.

Існують концепції побудови системи клієнт-сервер:

Слабкий клієнт – потужний сервер. У такій моделі вся обробка інформації перенесена на сервер, а у клієнта права доступу суворо обмежені. Сервер відправляє відповідь, яка не вимагає додаткової обробки. Клієнт взаємодіє з користувачем: складає та відправляє запит, приймає результат і виводить інформацію на екран.

Сильний клієнт – концепція, в якій частина обробки інформації надається клієнтові. У такому випадку сервер виступає сховищем даних, а вся робота по обробці та подання інформації переноситься на комп'ютер клієнта.

Система (додаток), яка заснована на клієнт-серверній взаємодії, включає три основних компоненти: представлення даних, прикладний компонент, компонент управління ресурсами і їх зберігання.

Сервісно-орієнтована архітектура

Сервісно-орієнтована архітектура (SOA, service-oriented architecture) — це популярний архітектурний шаблон для створення програмних застосунків на основі окремих модулів. Асинхронною називають SOA, в якій кожен сервіс є автономним і виконує окреме завдання. Якщо синхронна система надсилає запит і чекає негайної відповіді, то в цьому дизайні комунікація між сервісами цього не потребує. Це означає, що клієнт може надіслати запит і перейти до інших завдань, не чекаючи відповіді.

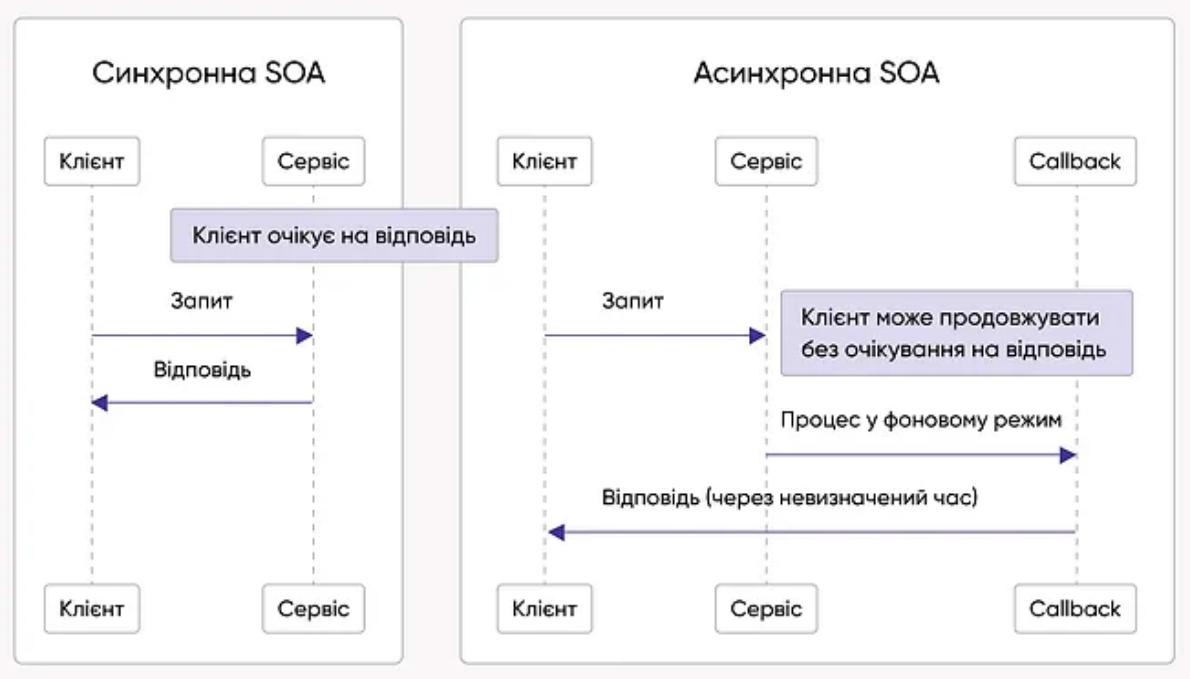

Рисунок 1. Синхронна/Асинхронна SOA

Основна одиниця SOA — сервіс, що відповідає за конкретний бізнес-процес. Наприклад, у додатку можуть бути окремі сервіси для:

- авторизації користувачів;
- логування;
- сповіщень.

Великі завдання поділяються між модулями. Для опису вмісту та функцій сервісу немає суворих правил. Команда сама визначає, що потрібне сервісу. При цьому самі сервіси дотримуються контрактів (чітких правил):

- що саме може підключитися до сервісу;
- за яким протоколом і через який інтерфейс йде передача даних;
- з якими даними працює сервіс;
- за які функції він відповідає;
- що сервіс повертає у відповідь на звернення.

Модулі нижнього рівня нічого не знають про реалізацію верхніх, крім протоколу, за яким вони працюють із ресурсами. Сервіси лише передають та приймають дані, не маючи доступу до методів один одного. Крім того, усі події обробляються асинхронно. Не повинно бути ситуацій, коли сервіси не працюють через те, що очікують відповіді від інших систем.

Оскільки сервіси представляють окремі функції, їх можна перевикористовувати в різних додатках. Об’єднання сервісів у більші сутності називається оркеструванням. Розробник отримує конструктор, з якого може збирати програми.

Компоненти взаємодіють між собою за протоколами за допомогою черги подій. Вона постачається через Enterprise Service Bus (ESB) — програмне забезпечення, яке керує передачею повідомлень між компонентами системи. Додаток, наступний SOA, умовно поділяється на шари, кожен із яких відповідає за певну роботу:

- *Business Process Layer* — шар для об’єднання сервісів та розв’язання завдань програми;
- *Service* — шар із сервісами;
- *Components* — шар із компонентами, які забезпечують роботу окремих сервісів. Наприклад, компонент не має своєї власної бази даних — він звертається до БД, якою користуються відразу кілька сервісів;

- *Integration Layer* — шар, що зв’язує між собою компоненти окремого модуля.

Мікросервісна архітектура

Мікросервісна архітектура (Microservices Architecture) – це підхід до розробки програмного забезпечення, в якому додаток складається з невеликих незалежних компонентів, які називаються мікросервісами. Кожен мікросервіс відповідає за обробку одного або кількох пов’язаних запитів або функцій, і може бути розроблений, випробуваний та розгорнутий незалежно від інших мікросервісів.

Мікросервісна архітектура дозволяє розробникам більш ефективно розвивати та масштабувати складні додатки, так як мікросервіси можуть бути розгорнуті та масштабовані незалежно один від одного. Крім того, цей підхід дозволяє розробникам використовувати різні технології та мови програмування для кожного мікросервісу, що дає можливість використовувати найбільш підходящі інструменти для кожної конкретної задачі.

Одним із викликів мікросервісної архітектури є необхідність управління багатьма різними мікросервісами та їх взаємодією між собою. Також важливо забезпечити безпеку та доступність всіх мікросервісів, а також підтримувати зв’язок між мікросервісами на необхідному рівні.

Хід виконання лабораторної роботи

У рамках цієї роботи буде реалізовано серверну частину з використанням ASP.NET Core Web API, а клієнтську частину — за допомогою фреймворку Vue.js.

Серверна частина відповідає за обробку запитів, бізнес-логіку та збереження даних у базі. Для цього використовується ASP.NET Core Web API, оскільки ця технологія надає інструменти для створення швидкого, масштабованого та безпечного серверного рішення з використанням RESTful API. У серверному проєкті передбачено:

- Налаштування аутентифікації та авторизації за допомогою JWT-токенів.
- Робота з базою даних через Entity Framework Core.
- Створення контролерів для обробки запитів від клієнта.

Клієнтська частина реалізується за допомогою Vue.js — JavaScript-фреймворку. Вона відповідає за взаємодію користувача із застосунком і передачу запитів до серверної частини через RESTful API. У клієнтському проєкті передбачено:

- Створення динамічних компонентів для роботи з даними.
- Використання Vuex для управління станом застосунку.
- Інтеграція Axios для здійснення HTTP-запитів до API сервера.

Розробка серверної частини веб-застосунку

1. Налаштування проєкту

Налаштування середовища:

- WebApplication.CreateBuilder(args) для створення конфігурації.
- app.Environment.IsDevelopment() для перевірки середовища.

Реєстрація сервісів:

- Використання методів-розширень для додавання сервісів.
- Налаштування Swagger через AddSwaggerGen.
- Налаштування CORS, аутентифікації, авторизації.

```
// Program Configuration
var builder = WebApplication.CreateBuilder(args);

// Реєстрація сервісів
builder.Services.AddCustomServices(builder.Configuration);
builder.Services.AddBackgroundServices();
builder.Services.AddCustomAuthentication(builder.Configuration);
builder.Services.AddCustomAuthorization();
builder.Services.AddCustomCors();
```

Рисунок 2. Налаштування середовища та реєстрація сервісів

Контролери:

- Налаштування контролерів із фільтрами (ProducesAttribute, ConsumesAttribute, AuthorizeFilter) для глобального управління поведінкою API.

Логування:

- Використання AddHttpLogging для включення HTTP-логування.

Побудова додатка (Build) і його конфігурація:

- UseSwagger і UseSwaggerUI для документування API.
- Використання UseCustomMiddlewares для додавання власних проміжних обробників.
- Підтримка статичних файлів (UseStaticFiles).
- Маршрутизація контролерів через MapControllers.

```
var app :WebApplication = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI(options =>
    {
        options.SwaggerEndpoint( url: "/swagger/v1/swagger.json",  name: "API v1");
    });
}

app.UseCustomMiddlewares();
app.UseStaticFiles();
app.MapControllers();
app.Run();
```

Рисунок 3. Побудова додатка (Build) і його конфігурація

2. Налаштування бази даних та ORM Entity Framework Core

Створення моделі бази даних:

Створимо папку Models і додамо класи, що представляють сутності. Наприклад, для сутності AppUser:

```
public class AppUser : IdentityUser<Guid>
{
    [Required]
    [StringLength(50)]
    30 usages
    public string FirstName { get; set; }
    [Required]
    [StringLength(50)]
    30 usages
    public string LastName { get; set; }
    [DataType(DataType.Date)]
    3 usages
    public DateTime BirthDate { get; set; }
    [Required]
    [StringLength(100)]
    3 usages
    public string Country { get; set; }
    [Required]
    [StringLength(100)]
    3 usages
    public string City { get; set; }
    16 usages
    public string ProfilePictureUrl { get; set; } =
        "https://dotlystorage.blob.core.windows.net/user-photos-container/default-photo.png";
    5 usages
    public string? RefreshToken { get; set; }
    5 usages
    public DateTime? RefreshTokenExpiration { get; set; }
    public ICollection<BaseTask> Tasks { get; set; } = new List<BaseTask>();
    public ICollection<TeamMember> TeamMemberships { get; set; } = new List<TeamMember>();
    1 usage
    public ICollection<UserProfileImage> ProfileImages { get; set; } = new List<UserProfileImage>();
    1 usage
    public ICollection<UserProfileImage> UploadedImages { get; set; } = new List<UserProfileImage>();
    public ICollection<ReportFile> UploadedReports { get; set; } = new List<ReportFile>();
}
```

Рисунок 4. Модель AppUser

Створення контексту бази даних:

Entity Framework – це відмінне ORM-рішення, що допомагає в автоматичному режимі пов'язувати прості класи C# із внутрішніми таблицями БД.

Для початку роботи з базою даних за допомогою фреймворку необхідно створити клас, який успадкує всі свої властивості від класу *Microsoft.EntityFrameworkCore.DbContext*.

```
public class AppDbContext : IdentityDbContext<AppUser, AppRole, Guid>
{
    anstxp
    public AppDbContext(DbContextOptions dbContextOptions) : base(dbContextOptions)
    {
    }
```

Рисунок 5. Контекст бази даних (AppDbContext)

Налаштування підключення до бази даних:

Налаштування підключення в контейнері служб:

```
services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(configuration.GetConnectionString( name: "ConnectionString")));
```

Рисунок 6. Підключення в контейнері служб

3. Конфігурація Identity

У Program.cs додамо конфігурацію для Identity, яка визначає, як створювати й автентифікувати користувачів:

```
services.AddIdentity<AppUser, AppRole>(options =>
    {
        options.Password.RequiredLength = 5;
        options.Password.RequireNonAlphanumeric = false;
        options.Password.RequireUppercase = false;
        options.Password.RequireLowercase = false;
        options.Password.RequireDigit = false;
        options.Password.RequiredUniqueChars = 3;
    })
    .AddEntityFrameworkStores<AppDbContext>()
    .AddUserStore<UserStore<AppUser, AppRole, AppDbContext, Guid>>()
    .AddRoleStore<RoleStore<AppRole, AppDbContext, Guid>>()
    .AddDefaultTokenProviders();
```

Рисунок 7. Конфігурація для Identity

Система готова працювати з користувачами (AppUser) і ролями (AppRole), забезпечуючи безпечну автентифікацію та зберігання паролів.

4. Налаштування аутентифікації через JWT

Додаймо конфігурацію для JWT:

```
"Jwt": {
  "Key" : "UAzEJ8TtkdexUYvV7iAlBZoy13uqc78k",
  "Issuer" : "http://localhost:5154/",
  "Audience": "http://localhost:5154/",
  "EXPIRATION_MINUTES": 10
},
"RefreshToken": {
  "EXPIRATION_MINUTES": 2100
}
```

Рисунок 8. Конфігурація для JWT

Реєстрація JWT-аутентифікації**:**

```
public static AuthenticationBuilder AddCustomAuthentication
    (this IServiceCollection services, IConfiguration configuration)
{
    return services.AddAuthentication( configureOptions: options =>
        {
            options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
            options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
        })
        .AddJwtBearer(options =>
        {
            options.TokenValidationParameters = new TokenValidationParameters
            {
                ValidateAudience = true,
                ValidAudience = configuration["Jwt:Audience"],
                ValidateIssuer = true,
                ValidIssuer = configuration["Jwt:Issuer"],
                ValidateLifetime = true,
                ValidateIssuerSigningKey = true,
                IssuerSigningKey = new SymmetricSecurityKey
                    (Encoding.UTF8.GetBytes(configuration["Jwt:Key"]))
            };
        }); // AuthenticationBuilder
}
```

Рисунок 8. Реєстрація JWT-аутентифікації

Тепер застосунок використовує JWT для автентифікації користувачів.

5. Реалізація JWT-сервісу

```
public class JwtService: IJwtService
{
    private readonly IConfiguration _configuration;
    anstxp
    public JwtService(IConfiguration configuration)
    {
        _configuration = configuration;
    }
```

Рисунок 9. Ініціалізація JWT-сервісу

```
public AuthResponse CreateJwtToken(AppUser user)
{
    DateTime expiration = DateTime.UtcNow.AddMinutes
        (Convert.ToDouble(_configuration["Jwt:EXPIRATION_MINUTES"]));
    Claim[] claims = new Claim[]
    {
        new Claim( type: JwtRegisteredClaimNames.Sub, user.Id.ToString()),
        new Claim( type: JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()), //  JWT unique id
        new Claim( type: JwtRegisteredClaimNames.Iat,
            new DateTimeOffset(DateTime.UtcNow).ToUnixTimeSeconds().ToString(),
            ClaimValueTypes.Integer64),
        new Claim( type: ClaimTypes.NameIdentifier, user.Email!),
        new Claim( type: ClaimTypes.Name, user.UserName!),
        new Claim( type: JwtRegisteredClaimNames.Email, user.Email!),
    };
    SymmetricSecurityKey key = new SymmetricSecurityKey
        (Encoding.UTF8.GetBytes(_configuration["Jwt:Key"]));
    SigningCredentials credentials = new SigningCredentials
        (key, algorithm: SecurityAlgorithms.HmacSha256);
    JwtSecurityToken tokenGenerator = new JwtSecurityToken(
        issuer: _configuration["Jwt:Issuer"],
        audience: _configuration["Jwt:Audience"],
        claims,
        expires: expiration,
        signingCredentials: credentials
    );
    JwtSecurityTokenHandler tokenHandler = new JwtSecurityTokenHandler();
    string token = tokenHandler.WriteToken(tokenGenerator);
    return new AuthResponse()
    {
        Token = token,
        Email = user.Email,
        UserName = user.UserName,
        Expiration = expiration,
        RefreshToken = GenerateRefreshToken(),
        RefreshTokenExpiration = DateTime.Now.AddMinutes
            (Convert.ToInt32(_configuration["RefreshToken:EXPIRATION_MINUTES"])) // DateTime
    };
}
```

Рисунок 10. Метод для створення JWT-токена

Функціонал JwtService:

- Реалізує логіку для створення та валідації JWT-токенів:
- Створення токенів: генерує токен з клеймами.
- Валідація токенів: перевіряє коректність токена, виданого сервером.
- Генерація refresh-токенів: для довгострокової автентифікації.

6. Реалізація контролера для роботи з користувачами

```
[AllowAnonymous]
2 usages  new *
public class AuthController : CustomControllerBase
{
    private readonly IAuthService _authService;

    1 usage  new *
    public AuthController(IAuthService authService)
    {
        _authService = authService;
    }

    [HttpPost]
    [Authorize( policy: "NotAuthorized")]
    [Route( template: "register")]
    1 usage  new *
    public async Task<ActionResult> Register([FromBody] RegisterRequest registerRequest)
    {
        var result = await _authService.RegisterUserAsync(registerRequest);
        if (!result.IsSuccess) return BadRequest(result.Errors);

        return Ok(result.Data);
    }

    [HttpPost( template: "login")]
    [Authorize( policy: "NotAuthorized")]
    1 usage  new *
    public async Task<ActionResult> Login([FromBody] LoginRequest loginRequest)
    {
        var result = await _authService.LoginUserAsync(loginRequest);
        if (!result.IsSuccess) return Unauthorized(result.Errors);

        return Ok(result.Data);
    }
```

Рисунок 10. Контролер AuthController

Створюємо метод для реєстрації нового користувача. Зберігаємо дані користувача в базі даних через UserManager.

Реалізовуємо метод для логіна, який перевіряє дані користувача й повертає JWT-токен.

Реалізація клієнтської частини (Vue.js)

1. Створення форми реєстрації:

```
<template>  Show component usages
  <div class="login-form-container">
    <form
      action="#"
      id="signup-form"
      novalidate
      @submit.prevent="handleSubmission"
    >
      <h3>Sign Up</h3>

      <span>First Name</span>
      <input
        type="text"
        name="firstName"
        class="box"
        placeholder="Enter your first name"
        v-model="user.firstName"
      />
      <div class="error-message">{{ msg.firstName }}</div>

      <span>Last Name</span>
      <input
        type="text"
        name="lastName"
        class="box"
        placeholder="Enter your last name"
        v-model="user.lastName"
      />
      <div class="error-message">{{ msg.lastName }}</div>

      <span>Username</span>
      <input
        type="text"
        name="username"
        class="box"
        placeholder="Enter your username"
        v-model="user.userName"
      />
      <div class="error-message">{{ msg.userName }}</div>
```

Рисунок 11. Форма для реєстрації користувача

Шаблон містить HTML-структуру форми з полями вводу, повідомленнями про помилки та кнопкою "Submit". Поля використовують двосторонній зв'язок (*v-model*) для синхронізації даних з об'єктом user.

Подія *@submit.prevent="handleSubmission"* блокує стандартну відправку форми і викликає кастомний метод обробки даних.

2. Валідація даних на стороні клієнта

```
<script>
import axios from "axios";
import router from "@/router";
import formHelper from "@/mixins/form-helper";

export default {  Show usages  new *
  mixins: [formHelper],
  data() {...},
  watch: {
    "user.firstName": function () {
      this.msg.firstName = this.validateName(this.user.firstName);
    },
    "user.lastName": function () {
      this.msg.lastName = this.validateName(this.user.lastName);
    },
    "user.userName": function () {
      this.msg.userName = this.validateName(this.user.userName);
    },
    "user.email": function () {
      this.msg.email = this.validateEmail(this.user.email);
    },
    "user.phoneNumber": function () {
      this.msg.phoneNumber = this.validatePhone(this.user.phoneNumber);
    },
    "user.password": function () {
      this.msg.password = this.validatePassword(this.user.password);
    },
    "user.confirmPassword": function () {
      this.msg.confirmPassword = this.validateConfirmPassword(
        this.user.password,
        this.user.confirmPassword
      );
    },
    "user.birthDate": function () {
      this.msg.birthDate = this.validateForEmpty(this.user.birthDate);
    },
    "user.country": function () {
      this.msg.country = this.validateForEmpty(this.user.country);
    },
    "user.city": function () {
      this.msg.city = this.validateForEmpty(this.user.city);
    },
  },
```

Рисунок 12. Валідація введених даних на стороні користувача

Кожен метод перевіряє валідність певного поля (наприклад, мінімальна довжина, формат email). Повертає порожній рядок у разі успіху або текст повідомлення про помилку.

Валідація проводиться на основі методів:

- Простий перевірки на формат (регулярні вирази).
- Логіка валідації включає:
  - Мінімальну довжину тексту.
  - Співпадіння паролів.
  - Формат email та номер телефону.
- Повідомлення про помилки відображаються поруч із полем.

3. Відправка HTTP POST-запиту на сервер

Метод registerUser:

```
registerUser() {
  const userData = {
    firstName: this.user.firstName,
    lastName: this.user.lastName,
    userName: this.user.userName,
    email: this.user.email,
    phoneNumber: this.user.phone,
    password: this.user.password,
    confirmPassword: this.user.confirmPassword, // Include confirmPassword
    birthDate: this.formatDate(this.user.birthDate),
    country: this.user.country,
    city: this.user.city,
    userType: this.user.userType, // Always send userType as 0
  };

  axios
    .post( url: "http://localhost:5253/api/Auth/register", userData)
    .then((response) => {
      console.log(response.data);
      router.push("/sign-in");
    })
    .catch((error) => {
      console.error(error);
      this.msg.signup =
        "Something went wrong, maybe you're already registered";
    });
},
```

Рисунок 13. Метод registerUser

Даний метод формує об'єкт userData зі значеннями з полів та використовує бібліотеку axios для відправки HTTP POST-запиту для надсилання даних на сервер.

API-ендпоінт */api/Auth/register* обробляє запити на реєстрацію.

4. Налаштування роутингу

Створюємо окремий файл router/index.js з конфігурацією маршруту для сторінки реєстрації. Роутер додається до основного Vue-додатку.

```
import { createRouter, createWebHistory } from "vue-router";
import SignUpView from "@/views/SignUpView/SignUpView.vue";

const routes : [{path: string, component: {ne... = [
  {
    path: "/sign-up",
    name: "Sign Up",
    component: SignUpView,
  },
];

const router : Router = createRouter( options: {
  history: createWebHistory(process.env.BASE_URL),
  routes,
});

export default router;  Show usages  anstxp
```

Рисунок 14. Налаштування роутингу

5. Налаштування Vue-додатку

У файлі main.js підключаються всі необхідні компоненти та створюється Vue-додаток:

```
import { createApp } from "vue";
import App from "./App.vue";
import router from "./router";
import store from "./store";

createApp(App).use(store).use(router).mount( rootContainer: "#app");
```

Рисунок 15. Налаштування Vue-додатку

Тестування розробленої архітектури

Перевіримо інтерфейс клієнта для вводу власних даних для реєстрації:

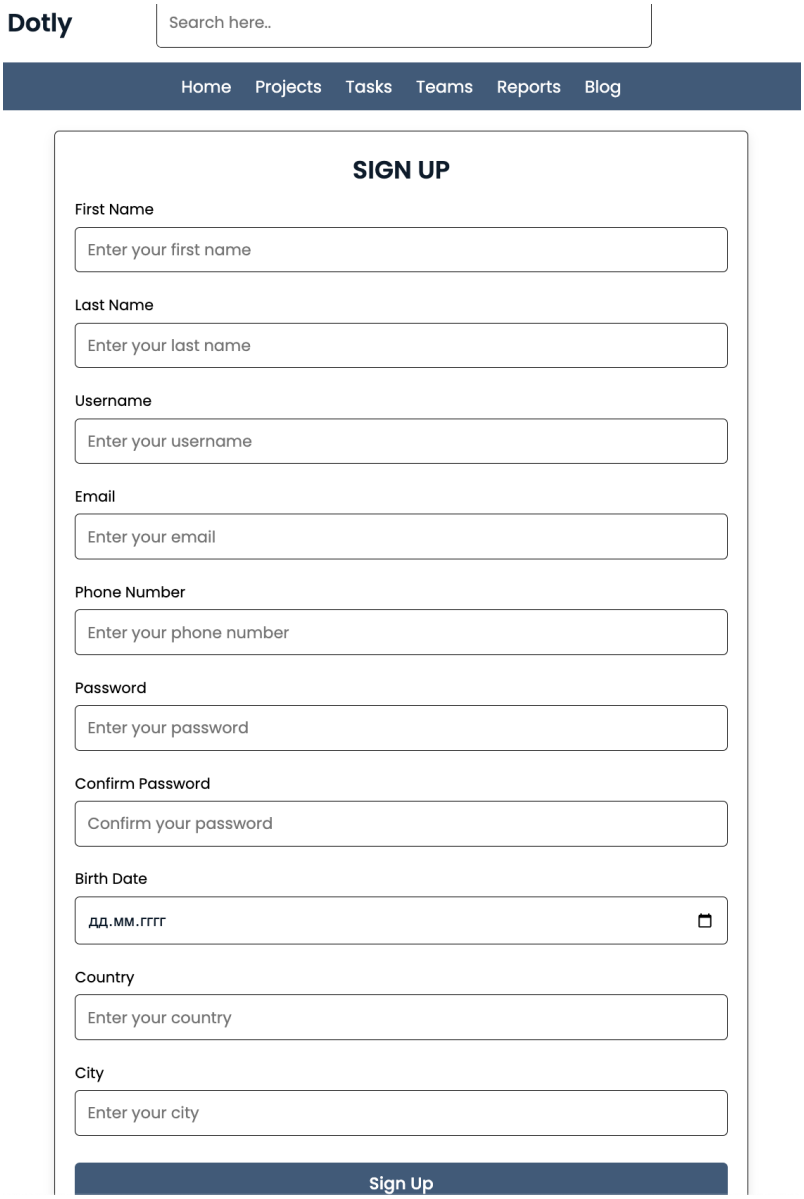

Рисунок 16. UI-інтерфейс для реєстрації

Перевіримо валідацію полів:

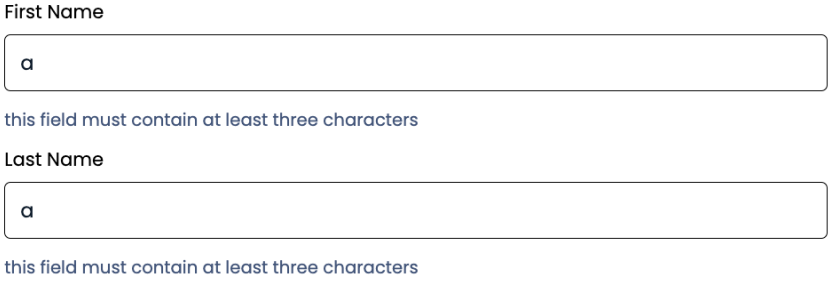

Рисунок 16. Валідація полів

Введемо тестові дані для перевірки надсилання даних за допомогою HTTP POST-запиту на сервер та отримання відповіді від сервера у вигляді JWT-токена:

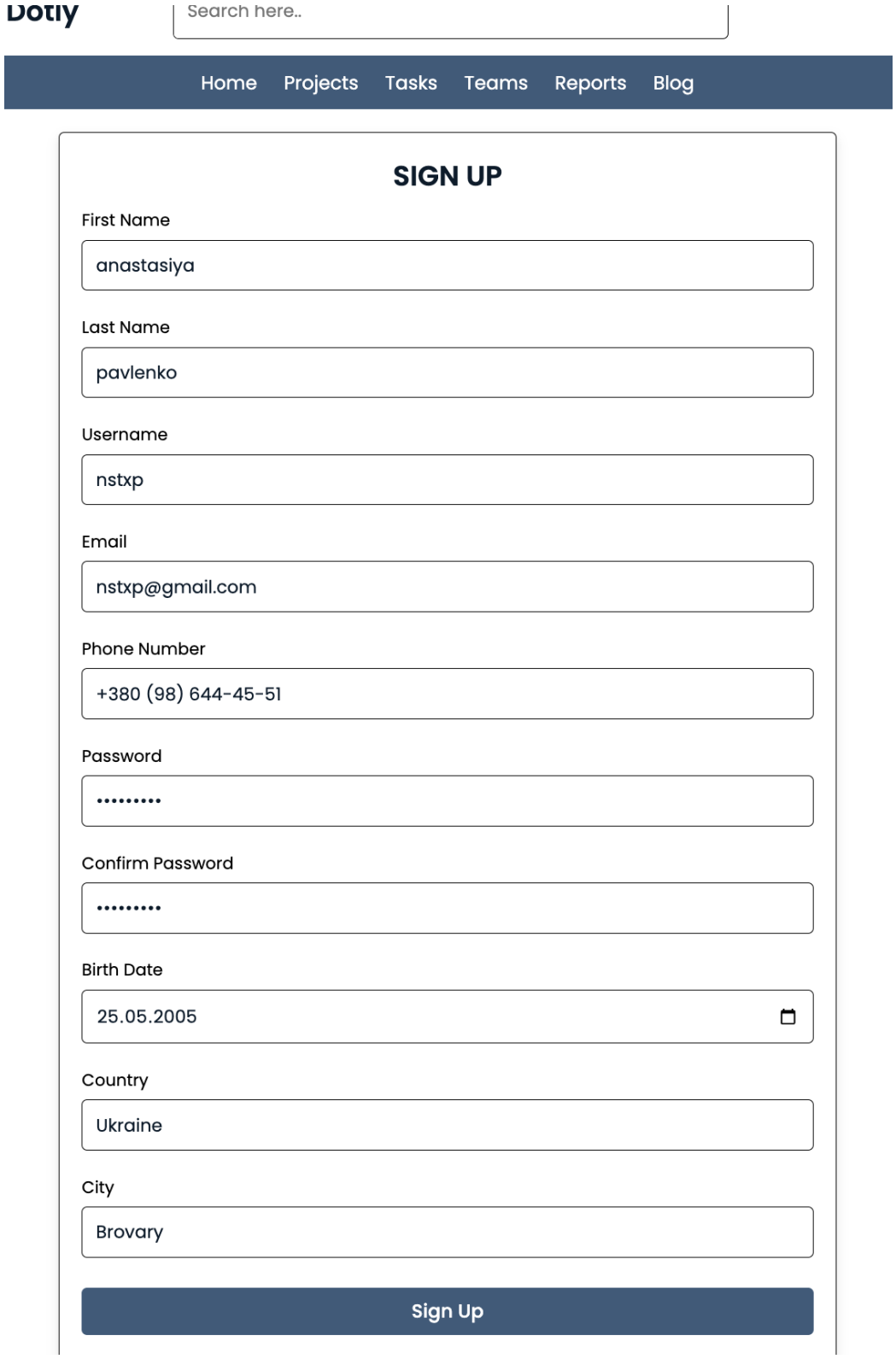

Рисунок 17. Перевірка за допомогою відправки тестових даних

Ми отримали відповідь від сервера, яка містить JWT-токен та refreshToken, отже, ми успішно виправили POST-запит на сервер:

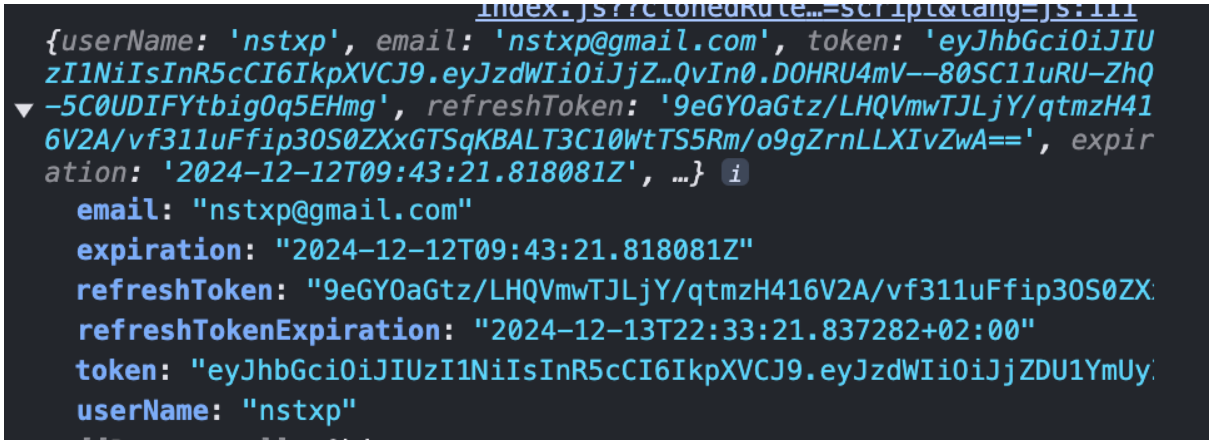

Рисунок 18. Відповідь сервера

Користувач може реєструватися та входити в систему через веб-інтерфейс.

Висновок

У результаті виконання лабораторної роботи було реалізовано програмне забезпечення для управління проектами в розподіленому середовищі, яке включає клієнтську та серверну частини, що взаємодіють через RESTful API. Використання технологій ASP.NET Core Web API для серверної частини та Vue.js для клієнтської дозволило забезпечити ефективну обробку запитів, масштабованість і зручний інтерфейс для користувачів.

На сервері була реалізована аутентифікація та авторизація за допомогою JWT-токенів, що дозволяє забезпечити безпеку доступу до даних. Завдяки використанню Entity Framework Core, дані ефективно зберігаються та обробляються у базі даних. Клієнтська частина забезпечує зручну взаємодію з користувачем через динамічні компоненти, управління станом за допомогою Vuex та інтеграцію з Axios для здійснення HTTP-запитів.