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»

Виконав:
студент групи ІА-34
Тунік О.

Перевірив:
асистент кафедри ІСТ
Мягкий М.Ю.

Київ 2025

Зміст

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

Теоретичні відомості

Клієнт-серверна архітектура

Клієнт-серверна архітектура це фундаментальна модель розподілених систем, де ролі жорстко розділені: клієнти ініціюють запити та відповідають за інтерфейс користувача, а сервери надають ресурси, обробляють бізнес-логіку та зберігають дані. Існує континуум реалізацій від «тонких» клієнтів (веб-браузери), які делегують майже всю логіку серверу, до «товстих» клієнтів (десктопні програми), що містять значну частину логіки та можуть функціонувати автономно. Ця модель є основою більшості сучасних мережевих застосунків.

Peer-to-Peer (P2P) архітектура

Архітектура «рівний-до-рівного» - це децентралізована модель, де кожен вузол (пір) одночасно виступає і клієнтом, і сервером, формуючи мережу рівноправних учасників. Замість централізованого контролю вузли безпосередньо обмінюються ресурсами (даними, обчислювальною потужністю), що забезпечує високу відмовостійкість та масштабованість, але водночас ускладнює керування безпекою, узгодженістю даних та ефективним пошуком ресурсів у великих мережах.

Сервіс-орієнтована архітектура (SOA)

Сервіс-орієнтована архітектура це модульний підхід, який представляє систему як набір слабко зв'язаних, перевикористовуваних сервісів з чітко визначеними інтерфейсами. Кожен сервіс інкапсулює специфічну бізнес-функцію та спілкується з іншими виключно через стандартизовані повідомлення (наприклад, через веб-служби SOAP/REST). Ключовою концепцією SOA є розділення уваги між повторним використанням сервісів та їх композицією для створення складних бізнес-процесів, часто з використанням шини сервісів (ESB) як посередника.

Мікросервісна архітектура

Мікросервісна архітектура стало еволюційним продовженням SOA. Воно полягає у розбитті додатку на набір найменших, автономних сервісів, кожен з яких відповідає за окрему бізнес-можливість. Кожен мікросервіс працює у власному процесі, має незалежний життєвий цикл розгортання та спілкується з іншими через легкі протоколи. Цей підхід робить систему високо масштабованою, гнучкою та стійкою до збоїв, оскільки дозволяє незалежно оновлювати та масштабувати окремі частини системи.

Хід роботи

Поставлене завдання

Flexible automation tool (strategy, command, abstract factory, facade, interpreter, SOA)

Інструмент автоматизації повинен забезпечувати найпростіші автоматичні дії для зручності користувача: завантаження нових фільмів / книг / файлів при випуску (наприклад, щоп'ятниці з'являються нові серії улюблених серіалів); встановити статуси в комунікаторах (skype – away при нульовій активності на тривалий час) і т.д. Автоматизація забезпечується шляхом введення правил (на зразок IFTTT.com сервісу), запису макросів (натискання клавіш, дії миші), планувальника завдань (о 5 ранку – початок роздачі торрент-файлів).

Реалізація архітектури

Реалізація сервіс-орієнтованої архітектури (SOA) в системі автоматизації FlexibleAutomationTool забезпечує фундаментальне відокремлення клієнтської логіки від серверних бізнес-сервісів, перетворюючи монолітну програму на набір незалежних, слабко зв'язаних служб, що спілкуються через стандартизовані контракти. Центральним елементом цієї трансформації виступає спільний проєкт FlexibleAutomationTool.Common, який містить інтерфейс IFlexibleAutomationService та об'єкт передачі даних RuleDto. Ці контракти формують незмінну угоду між клієнтом і сервером, де IFlexibleAutomationService чітко визначає доступні операції: отримання правил, їх виконання та управління, а RuleDto інкапсулює структуру даних, гарантуючи, що обидві сторони розуміють один одного незалежно від внутрішньої реалізації. Такий підхід до інтерфейсів як публічних контрактів є основою слабкого зв'язування, оскільки клієнтський код залежить лише від абстракції, а не від конкретних деталей сервера (рис. 1).

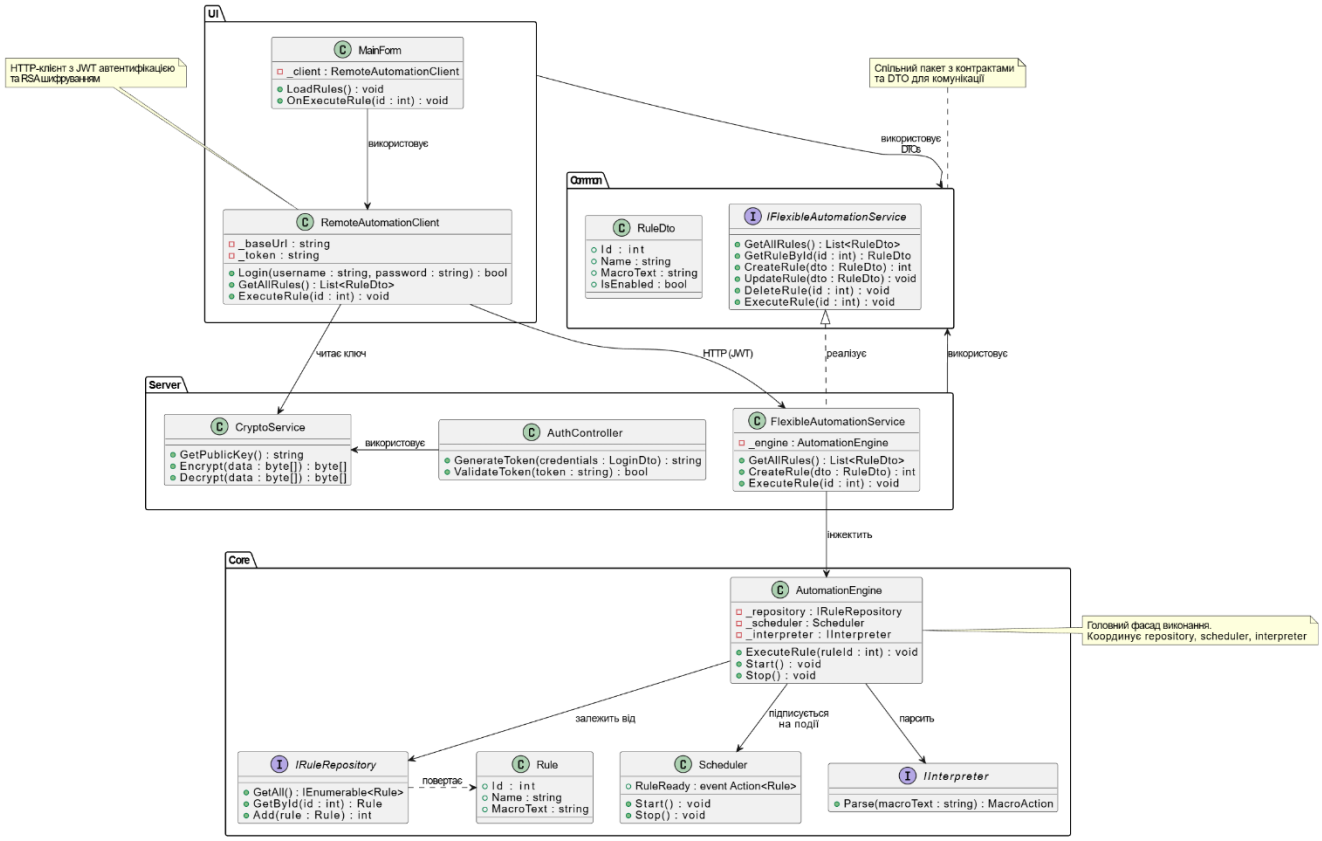

Рисунок 1. Діаграма класів архітектури Service-Oriented Architecture

Серверна частина, реалізована у проєкті FlexibleAutomationTool.Server як ASP.NET Core minimal API, виступає основним виконавцем цих контрактів. Вона надає RESTful ендпоїнти, такі як GET /api/rules та POST /api/rules/{id}/execute, які транслюють HTTP-запити у виклики методів реалізації IFlexibleAutomationService. Критично важливим для забезпечення безпеки в розподіленому середовищі є наявність ендпоїнів POST /token для видачі JWT та GET /api/crypto/publickey для надання RSA-ключа. Це реалізує модель безпеки, засновану на токенах для автентифікації та асиметричному шифруванні для конфіденційності переданих даних, наприклад, тіла запиту на виконання правила. InMemoryFlexibleAutomationService виконує роль демонстраційної реалізації сервісу, але її архітектурне призначення це бути заміненою на повноцінну службу, яка інжектить AutomationEngine та IRuleRepository з ядра системи. Саме таким чином серверний шар стає тонким фасадом, що делегує реальну бізнес-логіку до ядра (Core Layer), залишаючи за собою лише завдання комунікації, автентифікації та трансформації даних.

На стороні клієнта архітектурна цінність SOA матеріалізується в класі RemoteAutomationClient. Цей компонент абстрагує всю складність мережевої взаємодії: він отримує JWT-токен, керує його життєвим циклом, за потреби шифрує дані за допомогою отриманого від сервера публічного ключа та відправляє стандартизовані HTTP-запити. Для кінцевого користувача та UI-форм, таких як MainForm, цей клієнт виглядає як локальний сервіс, ніби всі операції відбуваються

в межах одного процесу. Форми викликають методи клієнта, як-от GetAllRules() або ExecuteRule(id), не турбуючись про те, що ці виклики трансформуються в HTTP-запити до віддаленого сервера. Це демонструє ключовий принцип SOA — інкапсуляцію та приховування складності розподіленої системи за простим інтерфейсом.

Загальна архітектурна вигода полягає в досягненні високої гнучкості та можливості розгортання. Серверний сервіс тепер може бути розміщений окремо: у хмарі, на локальному сервері або у вигляді контейнеризованого веб-сервісу. Кілька різних клієнтських застосунків (наприклад, Windows Forms, веб-інтерфейс або мобільний додаток) можуть одночасно використовувати один і той же набір служб через контракти з Common. Це розкриває потенціал для незалежного масштабування, оновлення та обслуговування компонентів. Отже, реалізована SOA не лише структурує код, але й закладає основу для еволюції системи у напрямку повноцінної мікросервісної архітектури, де кожен компонент, наприклад, CryptoService або сам AutomationEngine, міг би стати ще більш ізольованим та автономним сервісом.

Фрагменти коду реалізації

Контракт сервісу, загальний інтерфейс, який формалізує API між клієнтом і сервером:

```
namespace FlexibleAutomationTool.Common.Contracts
{
    public interface IFlexibleAutomationService
    {
        IEnumerable<RuleDto> GetRules();
        void ExecuteRule(int id, string? decryptedPayload = null);
    }
}
```

DTO правила. Спільна модель для передачі правил між компонентами:

```
namespace FlexibleAutomationTool.Common.Contracts
{
    public record RuleDto(int Id, string Name, string MacroText);
}
```

Endpoints minimal API — точки доступу сервера (JWT, отримання правил, виконання правила):

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

app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();

app.MapGet("/api/rules", (IFlexibleAutomationService svc) => Results.Ok(svc.GetRules()))
   .RequireAuthorization();

app.MapPost("/api/rules/{id}/execute", async (int id, IFlexibleAutomationService svc,
CryptoService crypto, HttpRequest req) =>
{
```

```
    using var sr = new StreamReader(req.Body, Encoding.UTF8);
    var body = await sr.ReadToEndAsync();
    string? decrypted = null;

    if (!string.IsNullOrEmpty(body))
    {
        if (body.StartsWith("ENC:", StringComparison.Ordinal))
        {
            var b64 = body.Substring(4);
            decrypted = crypto.DecryptBase64(b64);
        }
        else
        {
            decrypted = body;
        }
    }

    svc.ExecuteRule(id, decrypted);
    return Results.NoContent();
}).RequireAuthorization();

app.MapGet("/api/crypto/publickey", (CryptoService crypto) =>
{
    return Results.Ok(new { PublicKey = crypto.GetPublicKeyBase64() });
});

app.MapPost("/token", () =>
{
    var now = DateTime.UtcNow;
    var claims = new[] { new System.Security.Claims.Claim("sub", "demo-user") };
    var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtKey));
    var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);
    var token = new System.IdentityModel.Tokens.Jwt.JwtSecurityToken(
        issuer, audience, claims, expires: now.AddHours(8), signingCredentials: creds);
    var tokenString = new
System.IdentityModel.Tokens.Jwt.JwtSecurityTokenHandler().WriteToken(token);
    return Results.Ok(new { token = tokenString });
});

app.Run();
```

Клієнтський адаптер, приклад RemoteAutomationClient для виклику сервера, отримання публічного ключа і шифрування payload:

```
namespace FlexibleAutomationTool.UI.Services
{
    public class RemoteAutomationClient
    {
        private readonly HttpClient _http;

        public RemoteAutomationClient(HttpClient http, string? jwtToken = null)
        {
            _http = http ?? throw new ArgumentNullException(nameof(http));
            if (!string.IsNullOrEmpty(jwtToken))
                _http.DefaultRequestHeaders.Authorization = new
System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", jwtToken);
        }

        public async Task<IEnumerable<RuleDto>> GetRulesAsync()
        {
            var resp = await _http.GetFromJsonAsync<IEnumerable<RuleDto>>("/api/rules");
            return resp ?? Array.Empty<RuleDto>();
        }

        public async Task ExecuteRuleAsync(int id, string? sensitiveMacro = null)
```

```
        {
            HttpContent? content = null;
            if (!string.IsNullOrEmpty(sensitiveMacro))
            {
                try
                {
                    var pubResp = await _http.GetFromJsonAsync<Dictionary<string,
string>>("/api/crypto/publickey");
                    if (pubResp != null && pubResp.TryGetValue("PublicKey", out var
pubBase64) && !string.IsNullOrEmpty(pubBase64))
                    {
                        var cipher = EncryptWithServerPublicKey(pubBase64,
sensitiveMacro);
                        content = new StringContent("ENC:" +
Convert.ToBase64String(cipher), Encoding.UTF8, "text/plain");
                    }
                    else
                    {
                        content = new StringContent(sensitiveMacro, Encoding.UTF8,
"text/plain");
                    }
                }
                catch
                {
                    content = new StringContent(sensitiveMacro, Encoding.UTF8,
"text/plain");
                }
            }

            var resp = await _http.PostAsync($"/api/rules/{id}/execute", content ?? new
StringContent(string.Empty));
            resp.EnsureSuccessStatusCode();
        }

        private static byte[] EncryptWithServerPublicKey(string publicKeyBase64, string
plain)
        {
            var spki = Convert.FromBase64String(publicKeyBase64);
            using var rsa = RSA.Create();
            rsa.ImportSubjectPublicKeyInfo(spki, out _);
            var bytes = Encoding.UTF8.GetBytes(plain);
            return rsa.Encrypt(bytes, RSAEncryptionPadding.OaepSHA256);
        }
    }
}
```

Обґрунтування використання архітектури

Вибір сервіс-орієнтованої архітектури (SOA) для системи FlexibleAutomationTool є архітектурно обґрунтованим з огляду на реальні сценарії експлуатації та довгострокові вимоги до системи. У типовому корпоративному середовищі система автоматизації не існує ізольовано, вона повинна інтегруватися з іншими бізнес-системами, підтримувати різних клієнтів (десктопні, веб, мобільні застосунки) та забезпечувати централізоване управління правилами автоматизації для всієї організації. Без застосування SOA система залишалася б монолітним Windows Forms додатком, жорстко прив'язаним до конкретної платформи, що робило б неможливим її використання в сучасних гетерогенних IT-ландшафтах.

SOA дозволяє перетворити ядро системи – механізм виконання правил AutomationEngine – на універсальну службу, доступну через стандартизовані інтерфейси. Це критично важливо для сценаріїв, де автоматизаційні правила мають виконуватися на сервері, а не на клієнтських робочих станціях, наприклад, для обробки подій від корпоративних систем (CRM, ERP) або для виконання правил, що вимагають доступу до централізованих ресурсів. Реалізація через ASP.NET Core minimal API забезпечує легку інтеграцію з корпоративними системами за допомогою стандартних REST API, що є фактичним стандартом для міжсистемної взаємодії.

Застосування SOA також дозволяє чітко розділити відповідальності між різними командами розробників. Команда, що відповідає за ядро системи (Core Layer), може незалежно вдосконалювати механізми виконання правил та парсингу макросів, тоді як фронтенд-команда може розробляти різні клієнтські інтерфейси – від традиційних десктопних застосунків до сучасних веб-інтерфейсів або мобільних додатків. Спільний проєкт з контрактами (Common) виступає формалізованою угодою між цими командами, що зменшує ризики інтеграції та забезпечує узгодженість розробки.

Важливою перевагою використання SOA є можливість реалізації єдиної точки контролю безпеки та управління доступом. Централізований серверний сервіс дозволяє застосовувати єдині політики безпеки, вести централізоване логування всіх операцій виконання правил та контролювати доступ до функціоналу автоматизації на рівні користувачів або груп. Без SOA кожен клієнтський додаток містив би власну логіку безпеки, що призводило б до розходжень у реалізації та ускладнювало аудит.

Окремо слід відзначити архітектурну гнучкість, яку надає SOA. Система може поступово еволюціонувати від демонстраційної InMemory реалізації до повноцінного продуктивного рішення з підтримкою кластеризації, балансування навантаження та відмовостійкості. Клієнтські застосунки можуть адаптуватися до різних мережевих умов – від локального використання з сервером на тому ж комп'ютері до роботи з хмарною інфраструктурою, при цьому основна бізнес-логіка залишається незмінною.

Порівняно з альтернативними підходами, такими як монолітна архітектура або прямий доступ до спільної бази даних з різних клієнтів, SOA забезпечує значно кращу масштабованість та безпеку. Монолітний підхід обмежував би систему одним типом клієнта та ускладнював інтеграцію, тоді як прямий доступ до бази даних створював би проблеми з безпекою, узгодженістю даних та контролем доступу. SOA ж дозволяє створити чітку архітектурну межу між клієнтами та сервером, де сервер контролює всі операції з даними та бізнес-логікою, забезпечуючи цілісність та безпеку системи.

Застосування SOA також відкриває шлях для подальшої еволюції системи у напрямку мікросервісної архітектури. Такі компоненти, як CryptoService, AuthService або навіть окремі сервіси для різних типів правил, можуть бути виділені в незалежні мікросервіси з власним циклом розгортання та масштабування. Це робить архітектуру майбутньо-стійкою та готовою до вимог сучасних розподілених систем.

Враховуючи вимоги до безпеки, масштабованості, інтеграції та підтримки різноманітних клієнтів, SOA представляється оптимальним архітектурним вибором для FlexibleAutomationTool. Вона забезпечує баланс між складністю реалізації та архітектурними перевагами, дозволяючи створити систему, яка може ефективно функціонувати як в невеликих офісних середовищах, так і в великих корпоративних інфраструктурах.

Відповіді на контрольні запитання

1. Що таке клієнт-серверна архітектура?

Це модель розподіленої системи, де чітко розділені ролі: клієнти ініціюють запити, а сервери надають ресурси та обробляють ці запити.

2. Розкажіть про сервіс-орієнтовану архітектуру.

SOA – це архітектурний підхід, де система будується з набору незалежних, слабко пов'язаних сервісів, які спілкуються через стандартизовані інтерфейси та протоколи.

3. Якими принципами керується SOA?

SOA керується принципами слабкого зв'язування, повторного використання, автономності сервісів, стандартизації контрактів та можливості композиції сервісів.

4. Як між собою взаємодіють сервіси в SOA?

Сервіси взаємодіють виключно через обмін повідомленнями, зазвичай використовуючи стандартні протоколи, такі як HTTP з SOAP або REST.

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

Сервіси реєструються в реєстрі або каталозі служб (service registry), де розробники можуть знайти їх описи та контракти для подальшого використання.

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

Переваги: централізоване управління та безпека. Недоліки: залежність від сервера та ризик вузьких місць при високому навантаженні.

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

Переваги: децентралізація та відмовостійкість. Недоліки: складність управління, проблеми з безпекою та синхронізацією даних.

8. Що таке мікро-сервісна архітектура?

Це еволюція SOA, де система складається з дрібних, незалежних сервісів, кожен з яких відповідає за певну бізнес-можливість і має автономний життєвий цикл.

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

Для обміну даними використовуються легкі протоколи, такі як HTTP/REST, gRPC, AMQP або WebSockets.

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, це лише шаровий архітектурний патерн; SOA вимагає, щоб сервіси були розподіленими, автономними та доступними через стандартизовані мережеві інтерфейси.

Висновки

У ході виконання лабораторної роботи розглянули та реалізовали принципи сервіс-орієнтованої архітектури (SOA) у системі автоматизації FlexibleAutomationTool. Застосування SOA дозволило трансформувати монолітний додаток у розподілену систему з чітко відокремленими компонентами: клієнтським інтерфейсом, серверним API та спільним шаром контрактів. Реалізована структура з проєктами Common, Server та Core демонструє ефективне розділення відповідальностей та забезпечує слабке зв'язування між компонентами системи.

Отримані результати підтверджують доцільність використання SOA для систем, які потребують інтеграції з іншими бізнес-сервісами та підтримки різноманітних клієнтських застосунків. Реалізація централізованого серверного API на основі ASP.NET Core з JWT аутентифікацією та RSA шифруванням забезпечила необхідний рівень безпеки для розподіленого середовища. Клієнтський компонент RemoteAutomationClient ефективно інкапсулював складність мережевої взаємодії, надаючи зручний інтерфейс для UI-шару.

Архітектурне рішення дозволяє незалежно масштабувати та оновлювати серверну та клієнтську частини системи, що є критично важливим для корпоративних застосунків. Використання стандартизованих REST API та контрактів у проєкті Common створює основу для подальшої еволюції системи у напрямку мікросервісної архітектури. Загалом, реалізація SOA в межах лабораторної роботи підтверджує її ефективність для створення гнучких, масштабованих та легко інтегрованих програмних систем у сучасних гетерогенних IT-ландшафтах.