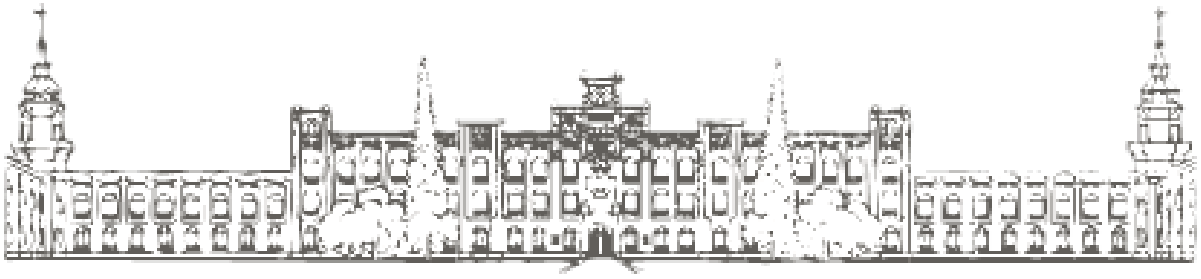

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

Звіт лабораторної роботи №9

з курсу

«Технології розроблення програмного забезпечення»

Виконавець:
Нікітченко Наталя Олегівна
студентка групи ІА-33
залікова книжка № ІА-3318

«13» 12 2025 р.

Перевірив: **Мягкий М. Ю.**

Київ – 2025

9. ЛАБОРАТОРНА РОБОТА № 9

Тема: Взаємодія компонентів системи.

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

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.

– Для однорангових мереж: реалізація взаємодії клієнтських додатків за допомогою WCF Peer to peer channel.

– Для SOA додатків: реалізація сервісу, що надає послуги клієнтським застосуванням; викладання сервісу в хмару або підняття у вигляді Web Service на локальній машині; використання токенів для передачі даних про автентифікації, двостороннє шифрування.

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.

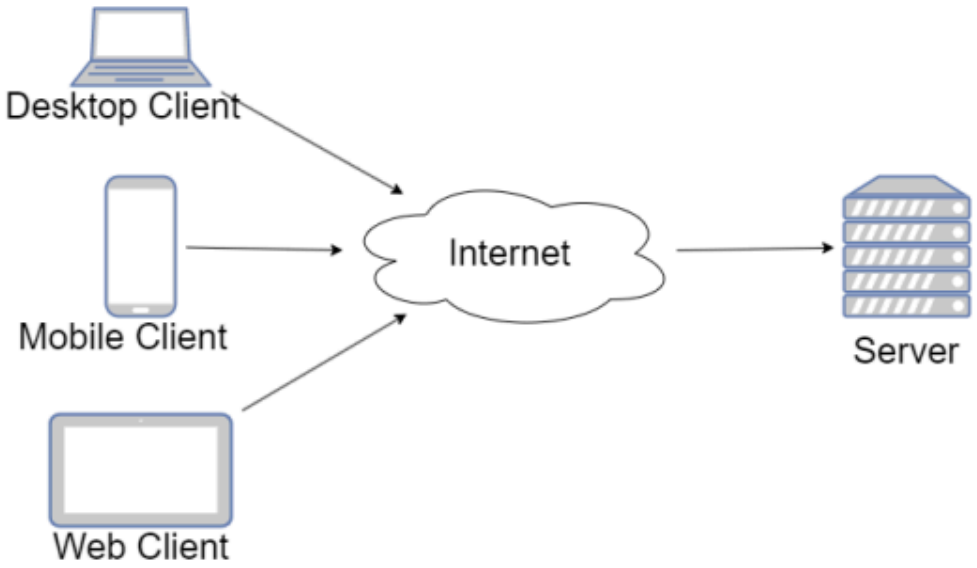

Рисунок 1. Клієнт-серверна архітектура

Товстий клієнт – антипод тонкого клієнта, більшість логіки обробки даних містить на стороні клієнта. Це сильно розвантажує сервер. Сервер в таких 100 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 101 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.

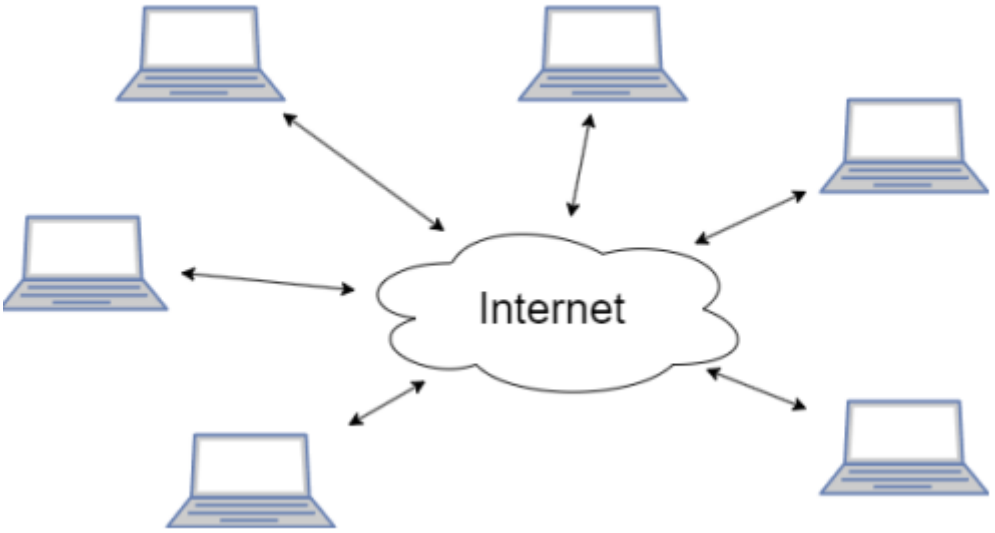

Рисунок 2. Peer-to-Peer архітектура

Основними сферами де peer-to-peer архітектура знайшла широке застосування є файлообмінники (BitTorrent), криптовалюти та інші блокчейн технології, 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.

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 Маклярті 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:

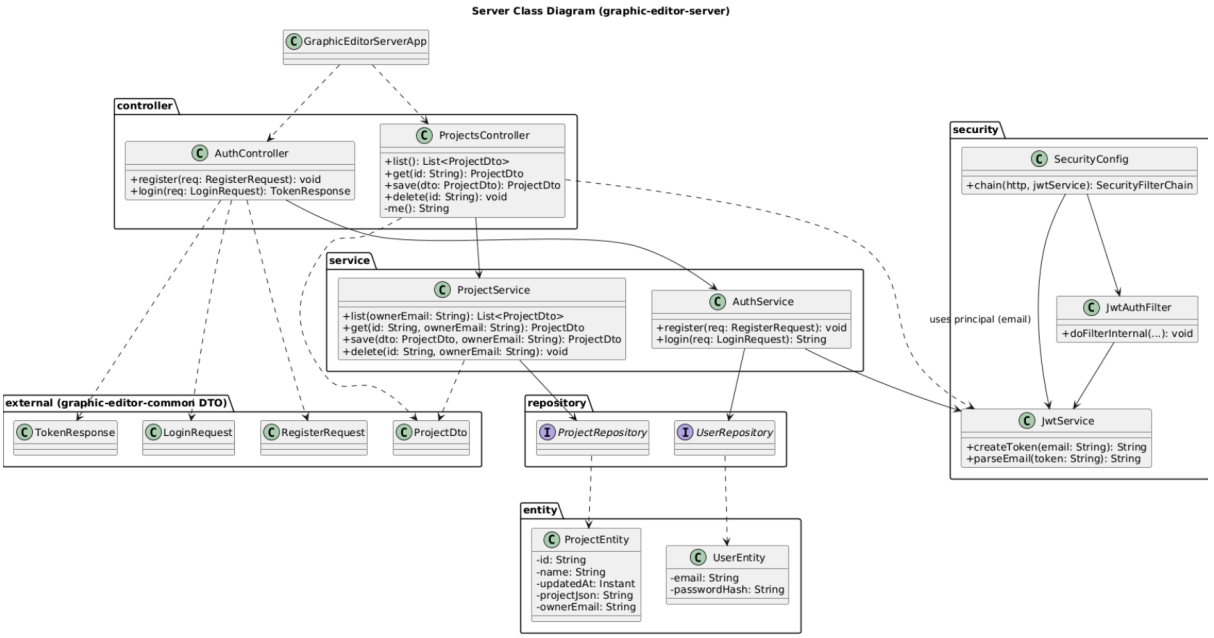

Рис.1 - Діаграма класів серверної частини застосунку

Діаграма відображає серверну частину графічного редактора у вигляді шарової архітектури: controller → service → repository → entity, а також окремий модуль security для JWT-автентифікації. У шарі controller розміщені AuthController і ProjectsController, які реалізують REST-ендпоінти та обробляють HTTP-запити/відповіді. Вся бізнес-логіка винесена в шар service: AuthService відповідає за реєстрацію, перевірку облікових даних і ініціацію створення токена, а ProjectService — за операції з проєктами (отримання списку, завантаження, збереження, видалення) з перевіркою власника.

Доступ до даних реалізований через repository (UserRepository, ProjectRepository), які працюють з JPA-сутностями entity (UserEntity, ProjectEntity) і забезпечують збереження інформації у базі даних. Для обміну між клієнтом і сервером використовуються DTO з модуля graphic-editor-common (LoginRequest, RegisterRequest, TokenResponse, ProjectDto), що формують контракт API та не прив’язані до структури БД.

Модуль security (SecurityConfig, JwtAuthFilter, JwtService) забезпечує захист ресурсів: фільтр зчитує JWT із заголовка Authorization, отримує email користувача та встановлює його як principal, після чого ProjectsController використовує цей principal для виконання операцій лише в межах даних поточного користувача. Така структура забезпечує розподілену взаємодію у стилі SOA: клієнт звертається до серверних сервісів через REST-інтерфейс, а сервер надає послуги з керування користувачами та проєктами.

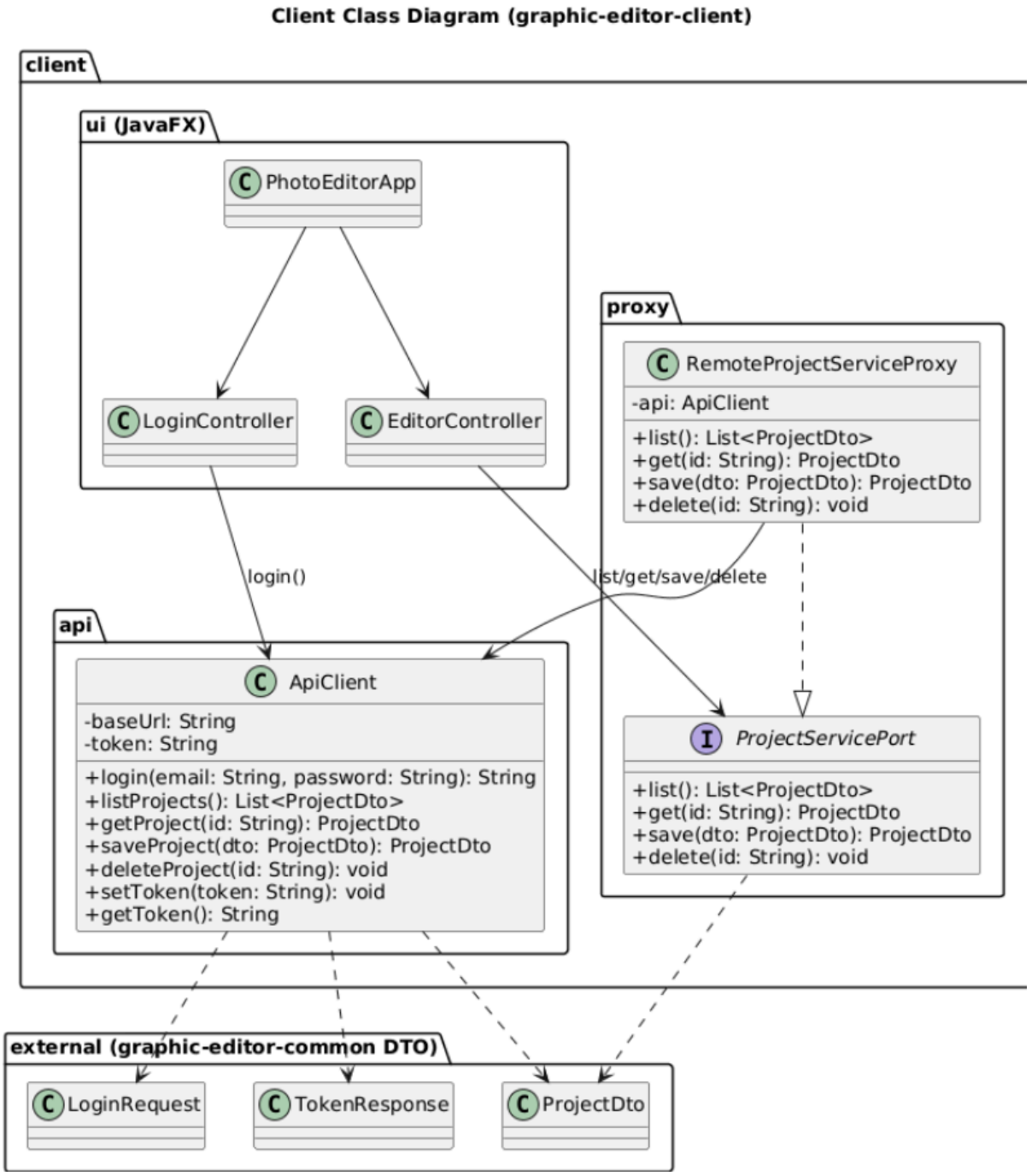

Рис.2 -Діаграма класів клієнтської частини

Діаграма показує клієнтську частину графічного редактора (JavaFX), яка працює з сервером по REST. У блоці ui (JavaFX) клас PhotoEditorApp запускає застосунок і використовує контролери LoginController та EditorController. Для мережевої взаємодії застосовується ApiClient, який виконує HTTP-запити до сервера: login, listProjects, getProject, saveProject,

deleteProject, а також зберігає JWT-токен (token) для авторизованих запитів.

Шар proxy реалізований класом RemoteProjectServiceProxy, який реалізує інтерфейс ProjectServicePort і виступає “посередником”: контролер редактора викликає методи list/get/save/delete, а проксі всередині делегує їх ApiClient. Для обміну даними використовуються DTO з модуля graphic-editor-common (LoginRequest, TokenResponse, ProjectDto), що утворюють контракт між клієнтом і сервером. Така структура відокремлює UI від деталей HTTP і робить клієнтську частину зручною для підтримки.

Фрагменти коду до цих частин:

```
package com.example.graphiceditor.server;
import jakarta.persistence.*;

@Entity
@Table(name = "users")
public class UserEntit {
    @Id
    private String email;
    private String passwordHash;

    public String getEmail() { return email; }
    public void setEmail(String email) {
this.email = email; }
    public String getPasswordHash() { return
passwordHash; }
    public void setPasswordHash(String
passwordHash) { this.passwordHash = passwordHash; }
}
```

```java
package com.example.graphiceditor.server;

import jakarta.persistence.*;
import java.time.Instant;

@Entity
@Table(name = "projects")
public class ProjectEntity {
    @Id
    private String id;

    private String name;

    private Instant updatedAt;

    @Lob
    @Column(columnDefinition = "CLOB")
    private String projectJson; // тут лежить весь ProjectDto як JSON

    private String ownerEmail;

    public String getId() { return id; }
    public void setId(String id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
```

```
        public Instant getUpdatedAt() { return
updatedAt; }
        public void setUpdatedAt(Instant updatedAt) {
this.updatedAt = updatedAt; }
        public String getProjectJson() { return
projectJson; }
        public void setProjectJson(String
projectJson) { this.projectJson = projectJson; }
        public String getOwnerEmail() { return
ownerEmail; }
        public void setOwnerEmail(String ownerEmail)
{ this.ownerEmail = ownerEmail; }
    }
    package com.example.graphiceditor.common.dto;

    public class AuthDtos {
        public record RegisterRequest(String email,
String password) {}
        public record LoginRequest(String email,
String password) {}
        public record TokenResponse(String token) {}
    }


    package com.example.graphiceditor.common.dto;

    import java.time.Instant;
    import java.util.List;
```

```
    public record ProjectDto(
            String id,
            String name,
            Instant updatedAt,
            List<LayerDto> layers
    ) {}


    package com.example.graphiceditor.repository;
    import
com.example.graphiceditor.server.ProjectEntity;
    import
org.springframework.data.jpa.repository.JpaRepositor
y;

    import java.util.List;

    public interface ProjectRepository1 extends
JpaRepository<ProjectEntity, String> {
       List<ProjectEntity>
findAllByOwnerEmailOrderByUpdatedAtDesc(String
ownerEmail);
    }
```

Висновки:

У ході виконання роботи було реалізовано функціонал графічного редактора в розподіленому середовищі у вигляді сервіс-орієнтованої взаємодії між клієнтом і сервером. Клієнтська частина (JavaFX) відповідає за інтерфейс користувача та роботу з полотном, а серверна частина (Spring

Boot) надає послуги керування даними проєктів через REST API. Для узгодженого обміну даними між компонентами створено спільний модуль з DTO, який формує контракт взаємодії та зменшує зв’язність між реалізаціями клієнта і сервера. Такий поділ дозволив чітко відокремити презентаційний шар від бізнес-логіки й доступу до даних, а також забезпечив можливість незалежного запуску, оновлення та масштабування серверної частини без змін у графічному інтерфейсі.

Безпека взаємодії була забезпечена використанням токенів JWT: після автентифікації користувач отримує токен, який додається до кожного подальшого запиту, а сервер перевіряє його валідність і встановлює контекст користувача (principal) для контролю доступу до ресурсів. Це обґрунтовано тим, що токенний підхід спрощує авторизацію у розподілених системах і не потребує збереження стану сесії на сервері, що підвищує масштабованість. Для логічного відокремлення мережевих викликів від UI на клієнті застосовано проксі-обгортку над сервісом роботи з проєктами, завдяки чому контролери працюють з інтерфейсом сервісу, а деталі HTTP-комунікації зосереджені в API-клієнті. Збереження даних реалізовано через репозиторії та сутності, що забезпечує структурований доступ до бази даних і можливість розширення моделі зберігання в подальшому.

Отриманий результат підтвердив, що обрана архітектура відповідає вимогам роботи в розподіленому оточенні: клієнт і сервер взаємодіють через стандартизований API, дані передаються у визначеному форматі, а доступ до операцій захищено механізмом автентифікації та авторизації. Практично було показано переваги SOA-підходу для такого класу застосунків: гнучкість розвитку системи, зрозумілий поділ відповідальностей і спрощення інтеграції з іншими клієнтами або сервісами. Перспективним напрямом удосконалення є розширення

функцій експорту/імпорту, деталізація зберігання об’єктів у БД на рівні окремих сутностей, а також повноцінне використання TLS/сертифікатів у середовищі розгортання для посилення захисту каналу зв’язку.

Контрольні питання:

1) Клієнт-серверна архітектура — це модель, у якій система поділена на дві ролі: клієнт ініціює запити та відображає результат, а сервер приймає запити, виконує обробку (логіку, доступ до даних) і повертає відповіді. Зазвичай сервер є централізованою точкою, що надає ресурси/послуги багатьом клієнтам через мережу.

2) Сервіс-орієнтована архітектура (SOA) — це підхід, коли функціональність системи оформлюється як набір незалежних сервісів, які надають “бізнес-можливості” через стандартизовані інтерфейси. Клієнти або інші сервіси взаємодіють із ними через мережеві протоколи, не знаючи внутрішньої реалізації, а лише контракт (API).

3) SOA керується принципами: слабке зв’язування (мінімальна залежність між сервісами), чіткі контракти (формально описані інтерфейси та формати даних), автономність сервісів (кожен може розгортатися/оновлюватися окремо), повторне використання (сервіс може обслуговувати різні клієнти), абстракція (сховати внутрішню реалізацію), композиція (складання бізнес-процесів із кількох сервісів), стандартизація взаємодії та керованість (моніторинг, політики безпеки).

4) Сервіси в SOA взаємодіють через мережу за принципом “запит-відповідь” (синхронно) або через повідомлення/черги (асинхронно). Типово це REST/SOAP/gRPC для синхронних викликів або

брокери повідомлень (RabbitMQ/Kafka) для подій і команд. Взаємодія відбувається через публічні контракти, а не через прямі виклики класів.

5) Про існуючі сервіси розробники дізнаються через документацію та реєстри сервісів. На практиці використовують OpenAPI/Swagger (опис REST API), WSDL (для SOAP), API-портали, service registry/discovery (Eureka, Consul), а також конфігурації середовищ (URL/ендпоінти). Запити роблять через HTTP-клієнти, SDK, згенеровані клієнти (наприклад, із OpenAPI), або через бібліотеки gRPC/SOAP.

6) Переваги клієнт-серверної моделі полягають у централізованому керуванні даними та безпекою, простішому адмініструванні, можливості контролю доступу та резервного копіювання на сервері, а також у тому, що клієнти можуть бути “тонкими” і не містити важкої логіки. Недоліки — залежність від доступності сервера (єдина точка відмови без резервування), потенційні “вузькі місця” при навантаженні, потреба масштабування сервера, а також вимоги до мережевого каналу та затримок.

7) Переваги однорангової (P2P) моделі в тому, що немає центрального сервера, вузли можуть напряму обмінюватися даними, система потенційно краще масштабується за рахунок ресурсів учасників і є стійкішою до відмов одного вузла. Недоліки — складніша безпека (довіра, автентифікація), важче забезпечити узгодженість даних, складніший пошук ресурсів/вузлів, проблеми з NAT/маршрутизацією, а також складніше адміністрування й моніторинг.

8) Мікросервісна архітектура — це варіант сервісного підходу, де система складається з багатьох невеликих сервісів, кожен відповідає за вузьку

бізнес-функцію, має власні дані (часто “своя” БД) і може розгортатися незалежно. Мікросервіси зазвичай дуже автономні, а команда може володіти сервісом “від коду до продакшену”.

9) Для обміну даними в мікросервісах найчастіше використовують HTTP/HTTPS (REST), gRPC поверх HTTP/2, інколи SOAP (рідше в сучасних системах), а для асинхронної взаємодії — протоколи/підходи через брокери повідомлень (AMQP для RabbitMQ, Kafka protocol), інколи MQTT для IoT. Формати даних: JSON, protobuf, Avro тощо.

10) Ні, такий підхід сам по собі не є SOA. Якщо між веб-контролерами та доступом до даних ти робиш “сервіси” як класи в межах одного застосунку, це просто шарова архітектура (controller → service → repository) і розділення відповідальностей всередині одного процесу. SOA починається тоді, коли “сервіс” є окремою мережею доступною одиницею (окремий процес/розгортання) з контрактом API, до якого звертаються по мережі інші компоненти/клієнти.