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

Лабораторна робота № 9

з дисципліни «**Технології розроблення програмного забезпечення**»

Взаємодія компонентів системи.

Виконав:

Студент групи ІА-31

Шереметьєв Дмитро

Київ – 2025

Вихідний код: https://github.com/Dimon4ick68/MindMapLabs

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.

Теоретичні відомості:

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.

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 (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 і 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 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 підвищення ефективності пошуку потрібно застосовувати спеціальні алгоритми.

Сервіс-орієнтована архітектура (SOA, англ. service-oriented

architecture) – модульний підхід до розробки програмного забезпечення, заснований на використанні розподілених, слабо пов’язаних сервісів або служб, оснащених стандартизованими інтерфейсами для взаємодії за стандартизованими протоколами. Історично сервіс-орієнтована архітектура з'явилася 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

Сама назва дає зрозуміти, що мікросервісна архітектура є підходом до створення серверного додатку як набору малих служб. 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.

Мікросервіси забезпечують чудові можливості супроводження у величезних комплексних системах з високою масштабуємістю за рахунок створення додатків, заснованих на множині незалежно розгортуючих служб з автономними життєвими циклами.

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.
- Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє спроєктовану архітектуру. Навести фрагменти програмного коду, які є суттєвими для відображення реалізованої архітектури.

Варіант:

20. **Mind-mapping software** (strategy, prototype, abstract factory, bridge, composite, SOA)

Візуальний додаток для складання "карт пам'яті" з можливістю роботи з декількома картами (у вкладках), автоматичного промальовування ліній, додавання вкладених файлів, картинок, відеофайлів (попередній перегляд); можливість додавання значків категорій / терміновості, обведення областей карти (поділ пунктирною лінією).

Хід роботи

У рамках виконання лабораторної роботи було спроєктовано та реалізовано взаємодію компонентів системи "MindApp" на основі архітектури **Клієнт-Сервер (Client-Server)** з використанням підходу **"Товстий клієнт" (Thick Client)**.

На відміну від підходу, де сервер генерує HTML-сторінки, у даній системі реалізовано чітке розділення відповідальності:

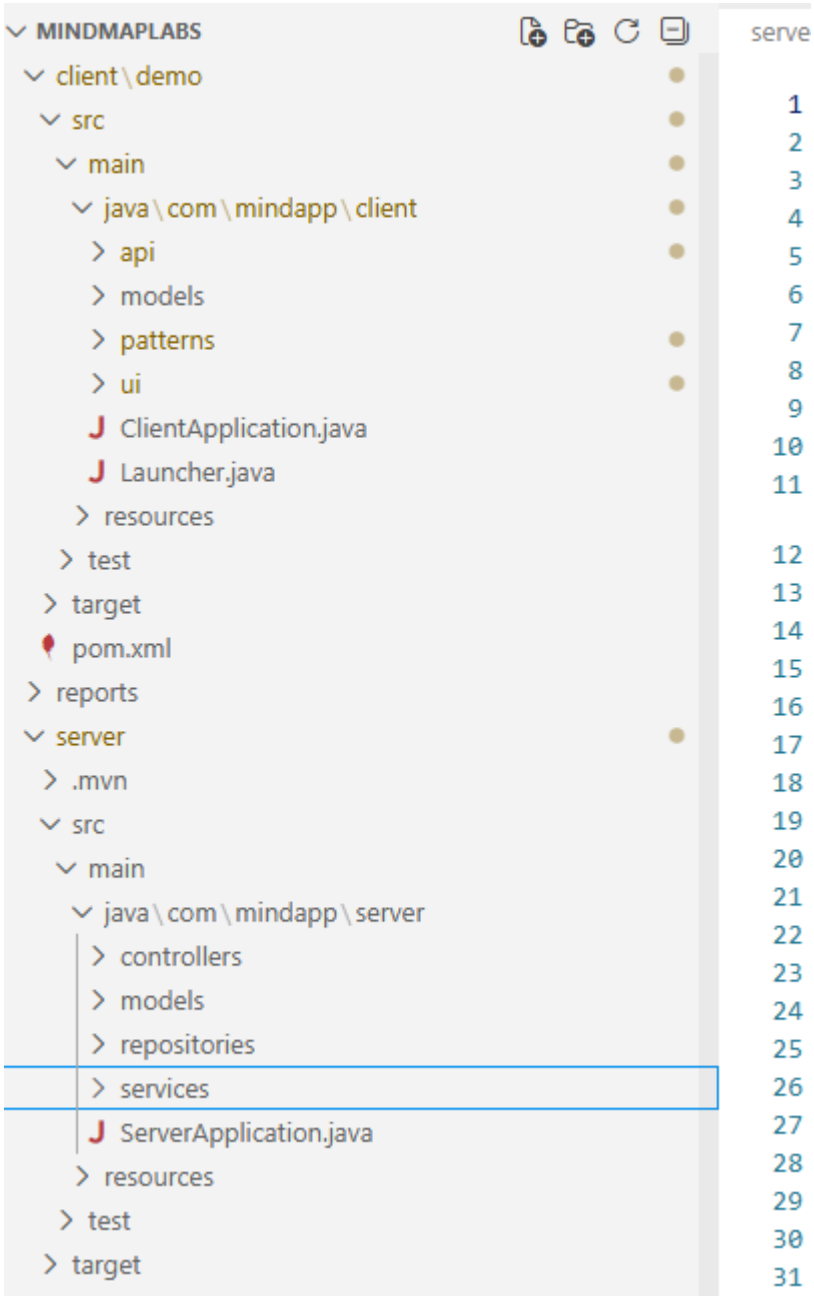

Рисунок 1 Архітектура проекту

1. **Клієнтська частина (Desktop Application):** Реалізована на **JavaFX**. Вона містить всю логіку відображення інтерфейсу, малювання ментальних карт, обробку подій миші та клавіатури. Клієнт є самостійним застосунком, який виконує обчислення (наприклад, рендеринг графіки) на стороні користувача.
2. **Серверна частина (REST API):** Реалізована на **Spring Boot**. Вона виступає виключно як постачальник даних та бізнес-логіки. Сервер не займається

візуалізацією, а надає **RESTful** інтерфейс для отримання та збереження даних у форматі **JSON**.

Реалізація взаємодії: Взаємодія між клієнтом та сервером відбувається через протокол **HTTP** за допомогою класу ApiClient на стороні клієнта. Цей клас інкапсулює всі мережеві запити, виступаючи фасадом для решти клієнтського коду.

- **Клієнт (ApiClient.java)**: Використовує java.net.http.HttpClient для відправки запитів.
  - GET /api/maps?userId=... — отримання списку карт.
  - POST /api/maps — збереження/оновлення карти (тіло запиту містить JSON-структуру карти).
  - DELETE /api/maps/{id} — видалення карти.
- **Сервер (MindMapController.java)**: Приймає ці запити, десеріалізує JSON у Java-об'єкти (MindMap, Node) та передає їх на шар сервісів.

Серверна архітектура (Layered Architecture): На стороні сервера застосовано багатошарову архітектуру, що забезпечує слабку зв'язність компонентів:

1. **Controller Layer (MindMapController):** Точка входу. Обробляє HTTP-запити, валідує вхідні дані та формує HTTP-відповіді.
2. **Service Layer (MindMapService):** Містить бізнес-логіку. Наприклад, перевіряє права доступу або готує дані перед збереженням.
3. **Repository Layer (MindMapRepository):** Відповідає за доступ до бази даних (**PostgreSQL**). Використовує Spring Data JPA для виконання SQL-запитів без написання їх вручну.

Така організація дозволяє клієнту працювати автономно, завантажуючи дані лише за потреби, що зменшує навантаження на мережу та сервер порівняно з "тонким клієнтом", який потребує перезавантаження сторінки при кожній дії.

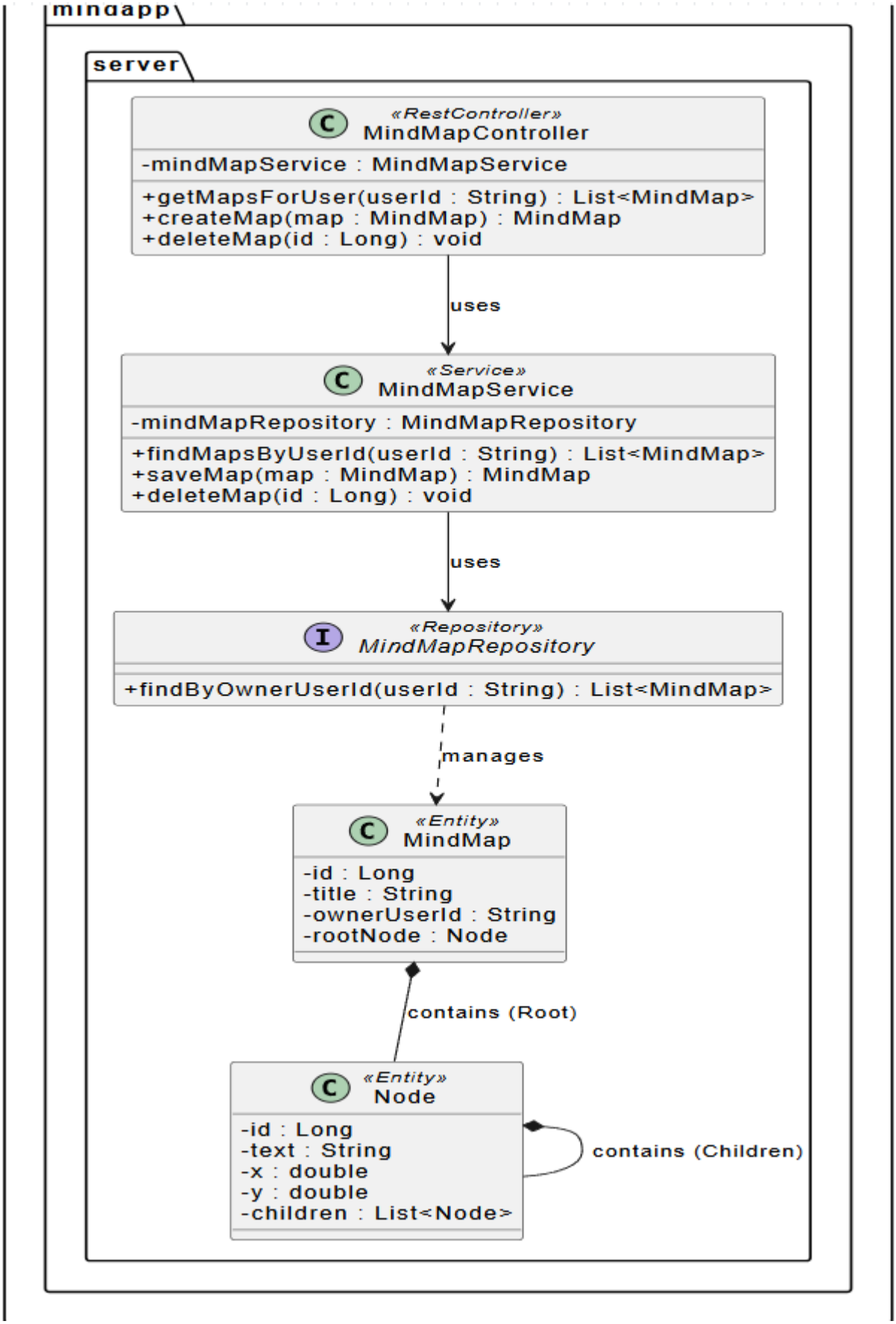

Рисунок 2. Діаграма класів серверу

На діаграмі зображено архітектуру серверної частини додатку, побудовану за патерном **MVC (Model-View-Controller)**, адаптованим для REST API (де View відсутня, оскільки дані повертаються у форматі JSON).

- **MindMapController:** Цей клас є вхідною точкою для HTTP-запитів від клієнта. Він визначає ендпоінти (URL-адреси) та методи обробки запитів (GET, POST, DELETE).
- **MindMapService:** Реалізує шар бізнес-логіки. Він інкапсулює правила обробки даних та виступає проміжною ланкою між контролером та репозиторієм.
- **MindMapRepository:** Інтерфейс, що розширює JpaRepository. Він забезпечує абстракцію доступу до даних, дозволяючи виконувати операції CRUD (Create, Read, Update, Delete) над сутностями без написання SQL-коду.
- **MindMap та Node:** Сутності (Entities), що відображають структуру бази даних. MindMap є кореневою сутністю, яка композиційно містить дерево вузлів Node.

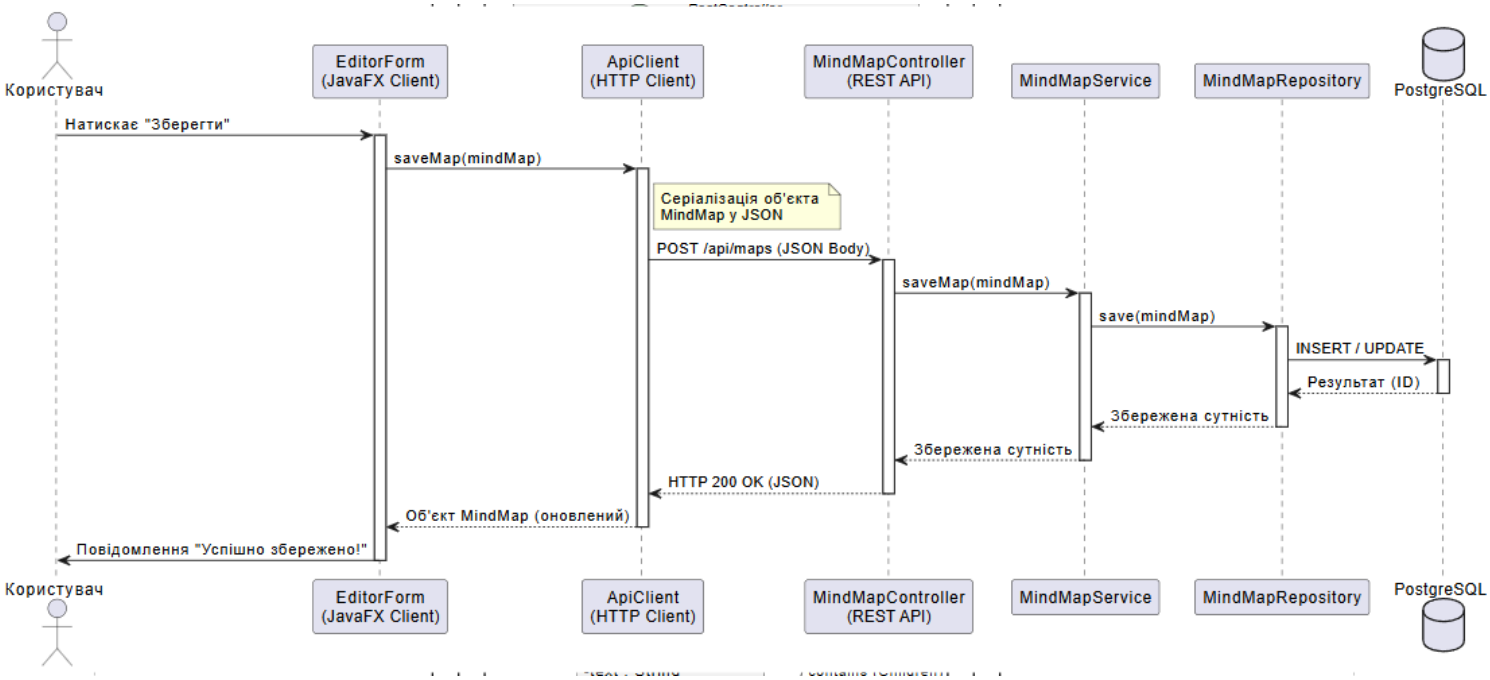

Рисунок 3 Діаграма послідовностей

Діаграма послідовності деталізує процес взаємодії компонентів системи при виконанні сценарію **"Збереження ментальної карти"**.

1. Процес починається на клієнті, коли користувач ініціює збереження.

2. Клієнтський клас ApiClient перетворює об'єкт карти у формат **JSON** та відправляє асинхронний **HTTP POST** запит на сервер.
3. MindMapController на сервері приймає запит та передає десеріалізований об'єкт у сервіс.
4. Сервіс викликає метод репозиторія save().
5. Репозиторій генерує відповідний SQL-запит до бази даних **PostgreSQL** для збереження або оновлення запису.
6. Після успішного виконання операції в БД, результат (оновлений об'єкт з ID) повертається по ланцюжку назад до клієнта, який відображає підтвердження користувачеві.

Приклад коду:

ApiClient.java

```java
package com.mindapp.client.api;

import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.mindapp.client.models.MindMap;

import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.List;


public class ApiClient {
    private static final String BASE_URL = "http://localhost:8081/api/maps";
    private final HttpClient client = HttpClient.newHttpClient();
    private final ObjectMapper mapper = new ObjectMapper(); // Для JSON

    // Отримати всі мапи користувача (GET)
    public List<MindMap> getMaps(String userId) throws Exception {
        HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(BASE_URL + "?userId=" + userId))
                .GET()
                .build();

        HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

        if (response.statusCode() == 200) {
            return mapper.readValue(response.body(), new TypeReference<List<MindMap>>(){});
        } else {
            throw new RuntimeException("Помилка завантаження: " + response.statusCode());
        }
    }
    public boolean register(String username, String password) {
        try {
            String json = String.format(format: "{\"username\":\"%s\", \"password\":\"%s\"}", username, password);
            HttpRequest request = HttpRequest.newBuilder()
                    .uri(URI.create(str: "http://localhost:8081/api/auth/register"))
                    .header(name: "Content-Type", value: "application/json")
                    .POST(HttpRequest.BodyPublishers.ofString(json))
                    .build();
            HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
            return response.statusCode() == 200;
        } catch (Exception e) { return false; }
    }

public boolean login(String username, String password) {
        try {
```

```java
public boolean login(String username, String password) {
        try {
            String json = String.format(format: "{\"username\":\"%s\", \"password\":\"%s\"}", username, password);
            HttpRequest request = HttpRequest.newBuilder()
                    .uri(URI.create(str: "http://localhost:8081/api/auth/login"))
                    .header(name: "Content-Type", value: "application/json")
                    .POST(HttpRequest.BodyPublishers.ofString(json))
                    .build();

            HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

            return response.statusCode() == 200;
        } catch (Exception e) { return false; }
    }
      You, 2 weeks ago • init commit, 1-3 lab
    public void deleteMap(Long id) throws Exception {
        HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(BASE_URL + "/" + id))
                .DELETE()
                .build();

        HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

        if (response.statusCode() != 200) {
            throw new RuntimeException("Помилка видалення: " + response.statusCode());
        }
    }

    // Зберегти мапу (POST)
    public MindMap saveMap(MindMap map) throws Exception {
        String json = mapper.writeValueAsString(map);

        HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(BASE_URL))
                .header(name: "Content-Type", value: "application/json")
                .POST(HttpRequest.BodyPublishers.ofString(json))
                .build();

        HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

        if (response.statusCode() == 200) {
            return mapper.readValue(response.body(), MindMap.class);
        } else {
            throw new RuntimeException("Помилка збереження: " + response.statusCode());
        }
    }
}
```

MindMapController.java

```java
package com.mindapp.server.controllers;

import java.util.List;...

@RestController
@RequestMapping("/api/maps")
public class MindMapController {

    @Autowired
    private MindMapService mindMapService;

    @GetMapping
    public List<MindMap> getMapsForUser(@RequestParam String userId) {
        return mindMapService.findMapsByUserId(userId);
    }

    @PostMapping
    public MindMap createMap(@RequestBody MindMap map) {
        return mindMapService.saveMap(map);
    }

    @DeleteMapping("/{id}")
    public void deleteMap(@PathVariable Long id) {
        mindMapService.deleteMap(id);
    }
}
```

You, 2 weeks ago • init commit, 1-3 lab

MindMapService.java

```java
You, 2 weeks ago | 1 author (You)
package com.mindapp.server.services;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.mindapp.server.models.MindMap;
import com.mindapp.server.repositories.MindMapRepository;

You, 2 weeks ago | 1 author (You)
@Service //
public class MindMapService {

    @Autowired //
    private MindMapRepository mindMapRepository;


    public List<MindMap> findMapsByUserId(String userId) {
        return mindMapRepository.findByOwnerUserId(userId);
    }

    public void deleteMap(Long id) {
        mindMapRepository.deleteById(id);
    }

    public MindMap saveMap(MindMap map) {
        return mindMapRepository.save(map);
    }
}

```

MindMapRepository.java

```java
package com.mindapp.server.repositories;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;

import com.mindapp.server.models.MindMap;



public interface MindMapRepository extends JpaRepository<MindMap, Long> {


    List<MindMap> findByOwnerUserId(String userId);
}       You, 2 weeks ago • init commit, 1-3 lab
```

Контрольні питання:

1. Що таке клієнт-серверна архітектура?

Клієнт-серверна архітектура виділяє два види додатків: клієнт та сервер. Клієнт представляє додаток користувачеві та дозволяє робити запити на сервер, який надає відповіді та реалізує бізнес-логіку. Клієнти бувають тонкими (всі операції обробки передають на сервер) та товстими (реалізує більшу частину обробки даних на стороні клієнта).

2. Розкажіть про сервіс-орієнтовану архітектуру.

Це архітектура, заснована на використанні розподілених, слабо пов’язаних сервісів або служб, оснащених стандартизованими інтерфейсами та протоколами для взаємодії. Сервіси взаємодіють між собою тільки за рахунок обміну повідомленнями, без створення спеціальних інтеграцій для доступу до однієї інформації, наприклад, до однієї бази даних.

3. Якими принципами керується SOA?

SOA керується такими принципами:
Слабка зв’язність - сервіси слабко пов’язані між собою.
Чітко визначена взаємодія - сервіси взаємодіють по чітко визначеним протоколам.
Повторне використання - сервіси можна повторно використати в різних системах.
Автономність сервісів - сервіси працюють незалежно та контролюють свій життєвий цикл.

4. Як між собою взаємодіють сервіси в SOA?

Сервіси взаємодіють по стандартизованих протоколах (http, smtp, ftp тощо) із використанням SOAP або REST інтерфейсів. Повідомлення між сервісами мають стандартні формати, наприклад json або xml.

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

Дізнатись про існуючий сервіс можна:

- Ручним пошуком у централізованому реєстрі сервісів.
- Динамічним виявленням сервіса у реєстрі.

Для того щоб зробити запит до сервісу потрібно знати його контракт. Найпоширенішими є SOAP та REST. Знаючи контракт розробник бачить якого формату запити потрібно надсилати: які необхідні параметри, які можливі помилки, які операції підтримуються.

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

Перевагою є простота розгортання, тому що оновлювати потрібно лише сервери і в результаті клієнти з наступними запитами автоматично будуть працювати з оновленою системою. Також централізовано зберігаються дані і реалізується безпека, сервери можна масштабувати.
Недоліками є залежність системи від сервера, потреба у постійному з’єднанні, підвищеному навантаженні на сервер.

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

Переваги:

Немає центрального сервера, кожен учасник рівноправний.

Краще масштабування.

Вища стійкість до збої через відсутність вузького місця у вигляді сервера.

Нижча вартість через відсутність потужного сервера.

Недоліки:

Складніше управляти безпекою.

Можливі проблеми з синхронізацією даних.

Кожен учасник мусить управляти ресурсами.

8. Що таке мікросервісна архітектура?

Вона є підходом до створення серверного додатку як набору малих служб. Більше орієнтована на серверну частину, де кожна служба виконується у своєму процесі і взаємодіє з іншими службами за такими протоколами, як HTTP/HTTPS, WebSockets чи AMQP.

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

Для обміну даними в мікросервісній архітектурі застосовуються протоколи HTTP/HTTPS (REST), gRPC, GraphQL, WebSocket, AMQP, MQTT та інші.

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, бо це проста реалізація багатошарової архітектури певного додатку. При сервіс-орієнтованій архітектурі кожний сервіс виступає окремим

додатком, який працює незалежно і може бути розгорнутим на окремому сервері, а також взаємодіє через мережу.