МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторних робіт з дисципліни
«АРХІТЕКТУРА І ТЕХНОЛОГІЇ ВЕБСЕРВІСІВ»

для магістрів спеціальності F7 «Комп’ютерна інженерія»
всіх форм навчання

Частина ІІ

2025

Методичні вказівки до виконання лабораторних робіт з дисципліни «Архітектура і технології вебсервісів» для магістрів спеціальності F7 «Комп’ютерна інженерія», всіх форм навчання. Частина ІІ / Укл. М.Б. Ільяшенко – Запоріжжя: НУ «Запорізька політехніка», 2025. – 64с.

Укладачі : М.Б. Ільяшенко, доцент, к.т.н.

Рецензент : Г.Г. Киричек, доцент, к.т.н.

Відповідальний
за випуск : М.Б. Ільяшенко, доцент, к.т.н.

Затверджено:
На засіданні кафедри
«Комп’ютерні системи та мережі»
Протокол № 2
від 29 серпня 2025 р.

Рекомендовано до видання
НМК факультету КНТ
Протокол № 2
від 10 вересня 2025 р.

ЗМІСТ

ВСТУП

Друга частина методичних вказівок з дисципліни «Архітектура і технології вебсервісів» присвячена основам побудови REST API з використанням фреймворку Spring Boot у поєднанні з реляційною базою даних MySQL. REST (Representational State Transfer) сьогодні є домінуючим стилем архітектури вебсервісів, що застосовується для створення легковагих, масштабованих і гнучких рішень. Його популярність пояснюється простотою, універсальністю та тісною інтеграцією з протоколом HTTP, що робить його зручним у використанні як для серверних, так і для клієнтських застосунків.

У процесі виконання лабораторних робіт студенти здобудуть знання про принципи REST, навчаться проєктувати ресурси та правильно будувати URI, опанують роботу з основними HTTP–методами (GET, POST, PUT, DELETE) для реалізації CRUD–операцій. Особлива увага буде приділена форматуванню запитів і відповідей у JSON, а також обробці типових кодів статусів, що є невід’ємною частиною взаємодії REST API з клієнтами.

Крім того, студенти отримають практичні навички інтеграції REST API з базою даних MySQL за допомогою Spring Data JPA. Це дозволить закріпити розуміння того, як сутності відображаються на таблиці в базі даних, як працюють репозиторії, що таке запити на основі імен методів, а також як реалізуються складні вибірки з використанням JPQL і нативних SQL–запитів.

Результатом виконання лабораторних робіт стане вміння створювати повноцінні серверні застосунки, здатні обробляти клієнтські запити та взаємодіяти з реальною базою даних. Такі навички є фундаментом для подальшого вивчення мікросервісної архітектури та складних сценаріїв інтеграції, які будуть розглядатися в наступній, третій частині методичних матеріалів.

3 ЛАБОРАТОРНА РОБОТА № 3 – ПОБУДОВА REST API ЗА ДОПОМОГОЮ SPRING BOOT

Мета роботи: одержати знання та навички створення REST API із використання фреймворку Spring Boot [1–7].

3.1 Теоретичні відомості

3.1.1 Що таке REST

REST (Representational State Transfer) – це архітектурний стиль взаємодії між клієнтом і сервером у розподілених системах, зокрема у веб–додатках. REST не є конкретним протоколом чи стандартом – це набір принципів і обмежень, які дозволяють створювати масштабовані, прості у використанні та ефективні веб–сервіси.

REST базується на стандартному протоколі HTTP і визначає правила того, як клієнт має надсилати запити та як сервер має відповідати. У REST–сервісах усі сутності (користувачі, замовлення, товари тощо) представляються як ресурси, які мають унікальні ідентифікатори (URI).

Наприклад:

GET /users/1 – отримати інформацію про користувача з ідентифікатором 1.

POST /orders – створити нове замовлення.

Таким чином, REST дозволяє будувати чіткі та зрозумілі API, які легко інтегрувати в різні системи.

6 принципів REST:

REST визначає кілька основних обмежень (принципів), дотримання яких робить API «RESTful».

1. Клієнт–сервер. У REST архітектура побудована за моделлю «клієнт–сервер». Це означає, що обидві частини системи мають чітко розділені обов’язки: клієнт відповідає за інтерфейс користувача та зручність взаємодії, а сервер – за

зберігання й обробку даних. Такий підхід дозволяє змінювати інтерфейс без втручання в серверну логіку та навпаки, що суттєво спрощує розвиток і масштабування системи.

2. Відсутність стану (Stateless). Кожен запит у REST є самодостатнім: сервер не зберігає інформації про попередні звернення клієнта. Це означає, що кожне повідомлення повинно містити всю необхідну інформацію для його виконання – наприклад, дані автентифікації чи параметри запиту. Такий принцип робить систему більш надійною, полегшує балансування навантаження між серверами та спрощує відновлення після збоїв.

3. Кешування. Щоб зменшити навантаження на сервери та прискорити обробку запитів, REST дозволяє використовувати кешування відповідей. Сервер може додавати до повідомлень спеціальні заголовки, які вказують клієнту, як довго відповідь вважається актуальною. Завдяки цьому часто запитувані дані можна зберігати локально, не звертаючись щоразу до сервера, а користувач отримує швидший відгук.

4. Єдиний інтерфейс. Однією з ключових ідей REST є уніфікований спосіб взаємодії між клієнтом і сервером. Це означає, що для всіх ресурсів використовуються стандартні HTTP–методи – GET, POST, PUT, DELETE та інші. Кожен ресурс має унікальний і зрозумілий ідентифікатор у вигляді URI. Завдяки цьому клієнтам не потрібно знати внутрішні деталі роботи сервера – достатньо дотримуватися загальних правил.

5. Багаторівнева система. REST допускає побудову архітектури у вигляді кількох рівнів. Клієнт не знає і не повинен знати, чи він взаємодіє безпосередньо з кінцевим сервером, чи з проміжним рівнем – наприклад, проксі, кешем або балансувальником навантаження. Це створює можливість прозорого масштабування, підвищує продуктивність і робить систему більш гнучкою та надійною.

6. Код за вимогою. Єдиний необов’язковий принцип REST – це можливість отримання коду за вимогою. У певних випадках сервер може надсилати клієнту виконуваний код, наприклад, JavaScript, що дозволяє розширити

функціональність додатка без необхідності оновлення клієнтської частини. Хоча цей принцип використовується рідше, він надає додаткову гнучкість і робить REST ще більш універсальним.

У лабораторних роботах важливо розуміти: щоб ваш API вважався RESTful, він має відповідати хоча б першим п’яти принципам. Шостий – додатковий і застосовується лише в окремих випадках.

REST став одним із найпоширеніших архітектурних стилів у веб–розробці завдяки своїй простоті, масштабованості та ефективності.

Переваги REST

1. Простота та зрозумілість. REST базується на стандартних HTTP – методах, таких як GET, POST, PUT і DELETE, що робить його дуже зрозумілим для розробників. Кожен запит має чітке призначення, а структура URI відображає логіку ресурсів. Це спрощує розробку та підтримку API, адже будь–який розробник, знайомий з HTTP, може швидко зрозуміти, як працює сервіс.

2. Незалежність клієнта та сервера. REST дозволяє клієнту та серверу розвиватися окремо. Клієнт може робити запити до сервера без знання його внутрішньої реалізації, а сервер може змінювати логіку або базу даних без потреби переписувати клієнтський код. Така незалежність робить систему більш гнучкою і масштабованою.

3. Масштабованість. REST API легко масштабувати, адже кожен запит є незалежним і не зберігає стан на сервері (stateless). Це дозволяє балансувати навантаження між кількома серверами і забезпечує стабільну роботу системи навіть при великій кількості користувачів.

4. Кешування та підвищена продуктивність. Завдяки принципу безстанності REST, відповіді сервера можуть кешуватися на стороні клієнта або проміжних проксі–серверах. Це зменшує кількість запитів до сервера, скорочує час відповіді і підвищує загальну продуктивність додатку.

5. Універсальність і стандартизація. REST використовує стандартизовані формати обміну даними, найчастіше JSON або XML, що забезпечує сумісність між різними платформами і мовами програмування. Завдяки цьому один і той

самий API може обслуговувати веб–клієнти, мобільні додатки та сторонні сервіси без додаткової адаптації.

6. Легка інтеграція з іншими сервісами. REST API дуже зручно інтегрувати з різними системами і мікросервісами завдяки простому протоколу HTTP. Взаємодія між сервісами відбувається через стандартизовані запити та відповіді, що спрощує побудову складних розподілених систем і дозволяє швидко додавати нові функції без порушення роботи існуючих.

3.1.2 Запити в REST та ресурси

У REST взаємодія між клієнтом і сервером відбувається за допомогою протоколу HTTP. Це означає, що будь–який REST–запит – це звичайний HTTP–запит, у якому використовується метод (GET, POST, PUT, DELETE тощо), URI (адреса ресурсу) та, за потреби, тіло запиту з даними.

URI як універсальна адреса ресурсу. Кожен ресурс у REST має власний унікальний ідентифікатор у вигляді URI (Uniform Resource Identifier). URI виконує роль «адреси», за якою можна отримати або змінити конкретний ресурс.

Приклади URI:

– /users – колекція всіх користувачів;

– /users/1 – конкретний користувач із ідентифікатором 1;

– /orders/25 – замовлення з номером 25.

Що може бути ресурсом? У REST ресурсом може бути будь–яка сутність предметної області, з якою працює система:

– користувачі (users);

– товари (products);

– замовлення (orders);

– повідомлення (messages);

– категорії (categories);

– навіть абстрактні поняття (наприклад, statistics, reports).

Іншими словами, ресурс – це будь–який об’єкт або набір об’єктів, якими можна керувати через API.

Правила іменування ресурсів. Щоб REST API був зрозумілим і послідовним, важливо дотримуватися кількох правил:

1. Використовувати іменники замість дієслів. URI описує ресурс, а не дію над ним. Наприклад, правильно писати /users/1, а не /getUser.

2. Колекції позначати у множині. Для групи ресурсів використовують множину: /users, а не /user.

3. Використовувати підресурси для вкладених структур. Якщо один ресурс містить інші, це відображається у шляху. Наприклад: /users/1/orders означає «усі замовлення користувача з ідентифікатором 1».

4. Використовувати нижній регістр і дефіс як роздільник. У назвах ресурсів використовують тільки малі літери; якщо потрібно розділити слова – застосовується дефіс. Наприклад: /product–categories.

5. Робити URI максимально простими й зрозумілими. Уникайте довгих описових шляхів. Краще написати /users, ніж /getAllUsersFromDatabase.

Приклади REST–запитів:

– GET /users → отримати список усіх користувачів;

– GET /users/5 → отримати дані користувача з id 5;

– POST /users → створити нового користувача (дані у тілі запиту);

– PUT /users/5 → оновити дані користувача з id 5;

– DELETE /users/5 → видалити користувача з id 5.

3.2.3 Тіло REST–запиту та формат JSON

У REST для передачі даних між клієнтом і сервером найчастіше використовується формат JSON (JavaScript Object Notation). JSON є простим текстовим форматом, який легко читається людиною та добре підтримується у більшості мов програмування.

Тіло (body) REST–запиту зазвичай містить JSON–об’єкт, який описує дані, що надсилаються на сервер. Це стосується в першу чергу методів POST і PUT, коли потрібно створити або оновити ресурс, приклади яких наведені в лістингах 3.1, 3.2 та 3.3.

Лістинг 3.1. – Створення нового користувача (POST /users)

```
{
  "name": "Іван Петренко",
  "email": "ivan.petrenko@example.com",
  "age": 28
}
```

У цьому запиті передаються дані нового користувача. Сервер створить ресурс і поверне відповідь із деталями (наприклад, призначить id).

Лістинг 3.2 – Оновлення інформації про користувача (PUT /users/1)

```
{
  "name": "Іван Петренко",
  "email": "ivan.p@example.com",
  "age": 29
}
```

У цьому випадку дані замінять попередню інформацію про користувача з ідентифікатором 1.

Лістинг 3.3 – Створення замовлення (POST /orders)

```
{
  "userId": 1,
  "items": [
    {
      "productId": 101,
```

Кінець лістинга 3.3

```
      "quantity": 2
    },
    {
      "productId": 205,
      "quantity": 1
    }
  ],
  "totalPrice": 350.50
}
```

Цей JSON описує замовлення: користувач із id = 1 купує два товари, з відповідною кількістю та загальною вартістю.

Важливо пам’ятати, що:

JSON будується на парах ключ–значення.

Ключі завжди беруться в лапки "...".

Значення можуть бути рядками, числами, булевими (true/false), масивами або вкладеними об’єктами.

Коди HTTP–відповідей у REST API. Коли клієнт надсилає запит до REST API, сервер завжди повертає HTTP–код відповіді, який показує результат обробки запиту. Це дозволяє клієнту зрозуміти, чи була операція успішною, чи виникла помилка, і яка саме.

Групи HTTP–кодів:

– 1xx – Інформаційні коди. Використовуються рідко. Повідомляють про те, що запит прийнято до обробки, але відповідь ще не завершена.

– 2xx – Успішні відповіді. Запит виконано успішно. Це найпоширеніша група у REST API.

– 3xx – Перенаправлення. Вказують, що ресурс переміщено або потрібно використати іншу адресу. У REST API зустрічаються нечасто.

– 4xx – Помилки з боку клієнта. Клієнт надіслав некоректний запит (невірний URI, неправильні дані, відсутня авторизація тощо).

– 5xx – Помилки з боку сервера. Сервер не зміг обробити правильний запит через внутрішню помилку.

Типові HTTP–коди для CRUD–операцій. У REST API CRUD–операції (Create, Read, Update, Delete) відповідають стандартним методам HTTP, більш детальніше про HTTP методи та Rest api наведено у таблиці 3.1. Нижче наведено рекомендовані коди:

1. Create (створення ресурсу) – POST:

– 201 Created – ресурс успішно створено. У відповіді зазвичай повертається тіло з новим об’єктом і заголовок Location із URI створеного ресурсу;

– 400 Bad Request – якщо надіслані некоректні дані.

2. Read (отримання ресурсу) – GET:

– 200 OK – успішне отримання ресурсу або колекції;

– 404 Not Found – ресурс із вказаним ідентифікатором не існує.

3. Update (оновлення ресурсу) – PUT або PATCH:

– 200 OK – успішне оновлення ресурсу, у відповіді можна повернути оновлені дані;

– 204 No Content – успішне оновлення без повернення тіла у відповіді;

– 400 Bad Request – якщо дані некоректні;

– 404 Not Found – якщо оновлюваного ресурсу не існує.

4. Delete (видалення ресурсу) – DELETE:

– 204 No Content – ресурс успішно видалено, і тіло відповіді не потрібне;

– 404 Not Found – якщо ресурс для видалення не знайдено.

Таким чином, у правильно побудованому REST API коди відповідей дають клієнту чітке розуміння результату запиту без необхідності аналізувати лише тіло відповіді.

Таблиця 3.1 – Відповідність між CRUD–операціями, HTTP–методами та типовими кодами відповідей у REST API

CRUD–операція	HTTP–метод	Опис	Типові коди відповідей
Create (створення)	POST	Створює новий ресурс у колекції	201 Created – ресурс створено 400 Bad Request – некоректні дані
Read (отримання)	GET	Повертає ресурс або список ресурсів	200 OK – успішно отримано 404 Not Found – ресурс не знайдено
Update (оновлення)	PUT / PATCH	Оновлює існуючий ресурс (повністю або частково)	200 OK – ресурс оновлено і повернуто 204 No Content – оновлено без тіла відповіді 400 Bad Request – некоректні дані 404 Not Found – ресурс не існує
Delete (видалення)	DELETE	Видаляє ресурс	204 No Content – ресурс видалено 404 Not Found – ресурс не знайдено

3.2 Хід роботи

Створення проєкту в IntelliJ IDEA Ultimate

Для створення нового проєкту виберіть у меню File → New → Project. У вікні, що відкрилося, зліва (у лівій колонці) виберіть Spring Boot і введіть параметри, як на рисунку нижче (систему збирання можна вибрати або Maven, або Gradle – на ваш вибір), як показано на рисунку 3.1.

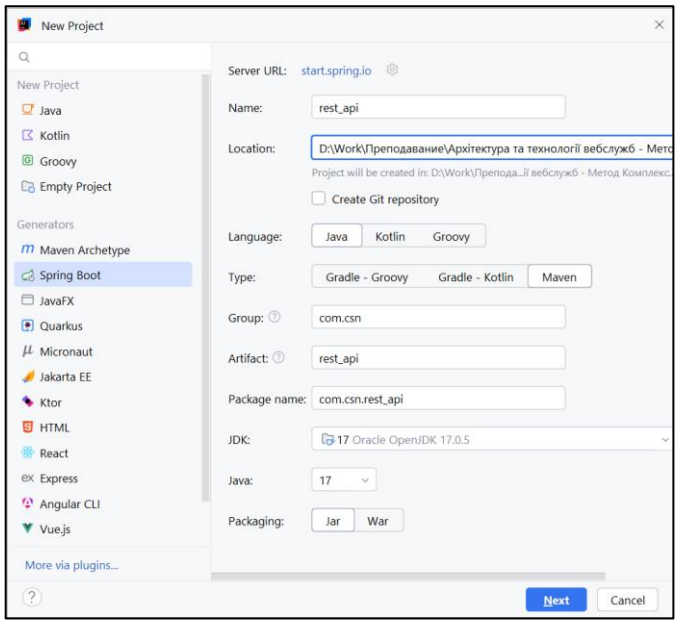

Рисунок 3.1 – Вікно налаштувань Spring Boot в IntelliJ IDEA

Натисніть Next, у наступному вікні виберіть пункт Web → Spring Web, як показано на рисунку 3.2.

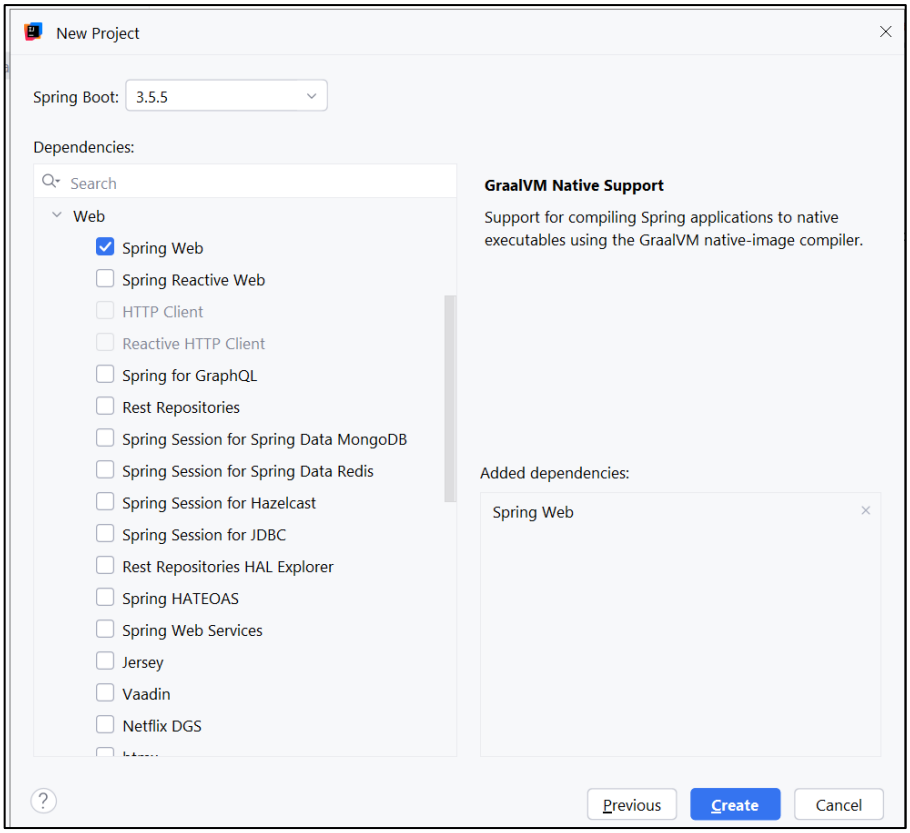

Рисунок 3.2 – Вікно налаштувань залежностей проекту

У результаті буде створено порожній Spring Boot застосунок зі структурою папок, як на рисунку 3.3.

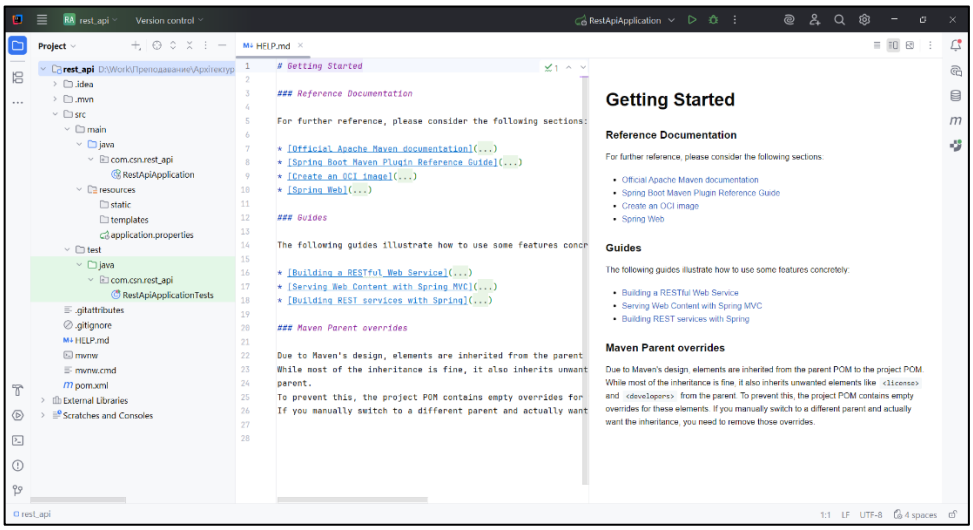

Рисунок 3.3 – Вікно новоствореного проекту Spring Boot

Застосунок уже має запускатися (Shift+F10) або іконка Run на верхній панелі. У результаті запуску ви повинні побачити такі повідомлення в лозі, як показано на рисунку 3.4.

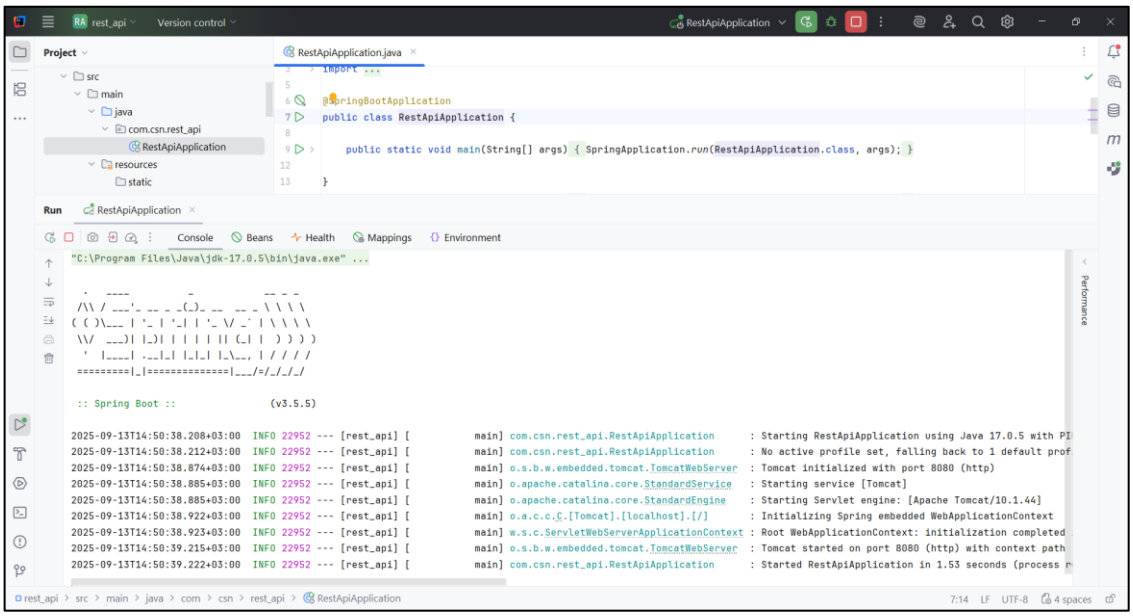

Рисунок 3.4 – Консольний вивід при вдалому запуску застосунка

У корені проєкту розташований файл pom.xml (у разі якщо була вибрана система збирання Maven), у якому в розділ <dependencies>… </dependencies> мають додатися такі залежності:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Також генератор повинен згенерувати код класу RestApiApplication, як наведено у лістингу 3.4.

Лістинг 3.4 – Код основного виконуваного класу застосунку

```
package com.csn.rest_api;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class RestApiApplication {

    public static void main(String[] args) {
        SpringApplication.run(RestApiApplication.class,
args);
    }

}
```

Створення проєкту в IntelliJ IDEA Community Edition. Для генерації конфігурації проєкту для системи збирання відкрийте сайт https://start.spring.io/ встановіть налаштування, аналогічні прикладу на рисунку 3.5.

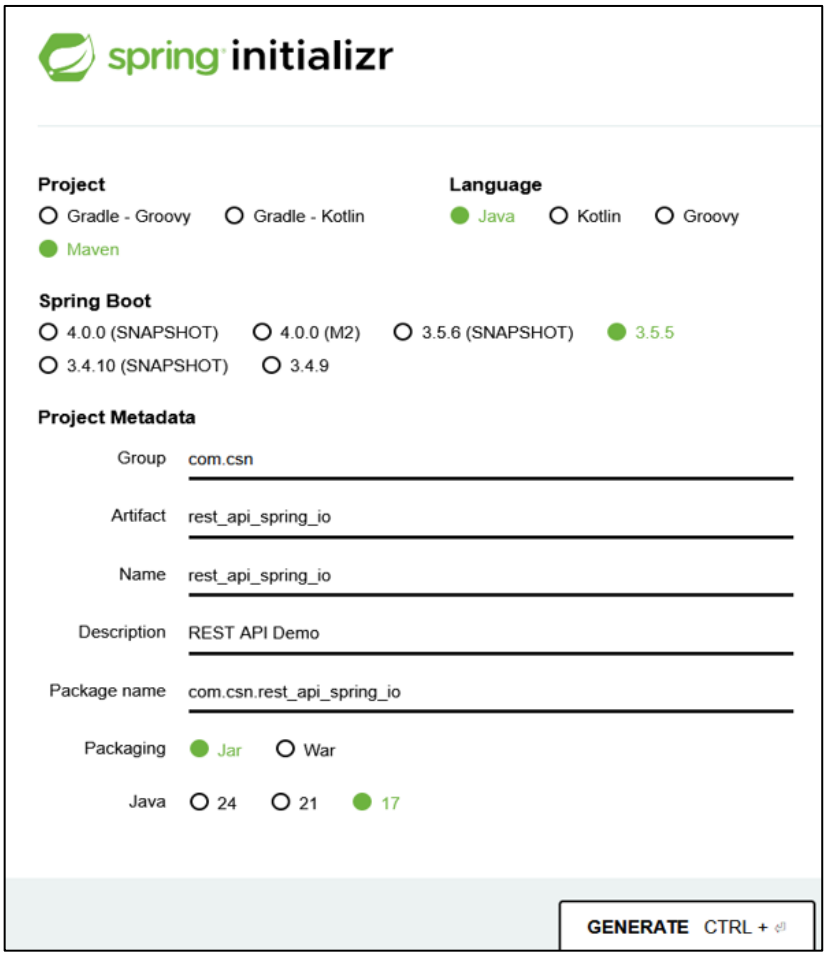

Рисунок 3.5 – Налаштування проекту в Spring Initializr

Натисніть Generate і збережіть архів на диску. Розпакуйте архів, після чого створіть новий проєкт у IntelliJ IDEA Community Edition, використовуючи меню File → New → Project from Existed Sources. Виберіть папку, у яку ви розпакували завантажений із сайту шаблон проєкту. У вікні, що з'явиться, виберіть систему збирання, яку ви вказали під час генерації шаблону проєкту (правильна система збирання має підставитися автоматично, але якщо ні – виставте правильну вручну), як показано на рисунку 3.6.

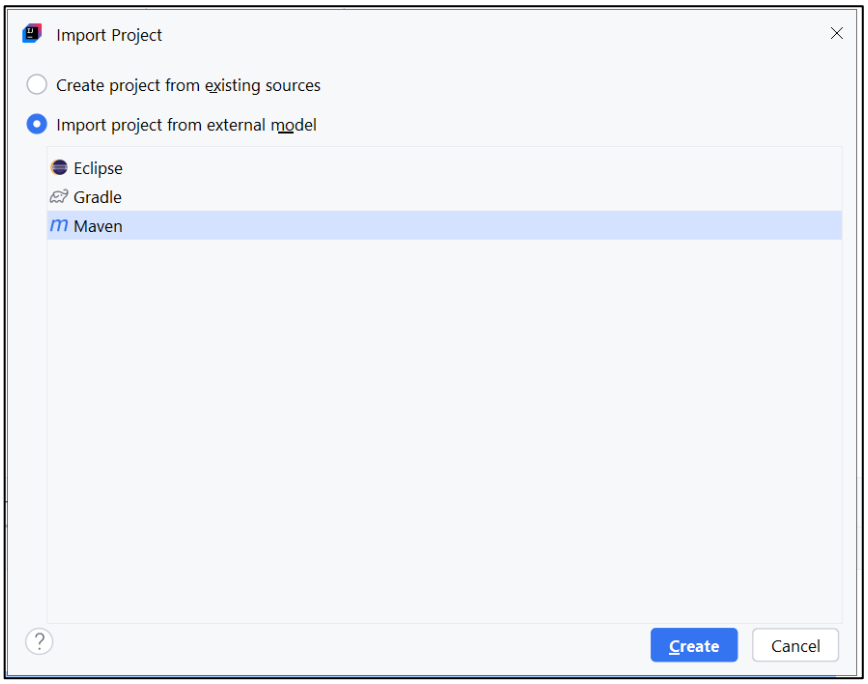

Рисунок 3.6 – Вінко налаштувань системи сборки проекту

Створіть новий Java–проєкт, вибравши File → New → Project. Виберіть налаштування, як на рисунку 3.7.

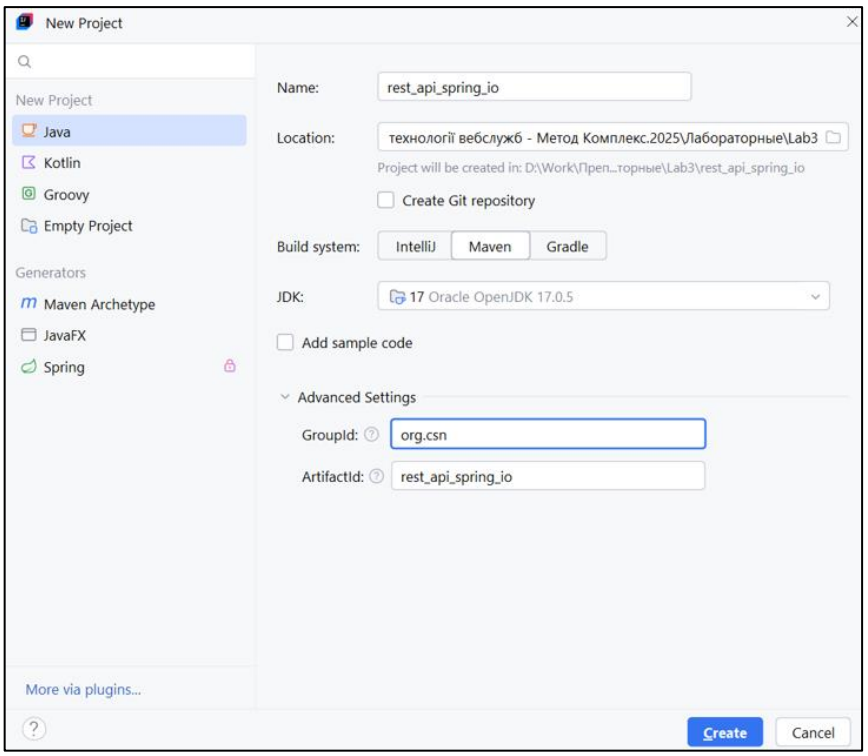

Рисунок 3.7 – Налаштування нового проекту

У результаті ви отримаєте проект, повністю аналогічний тому, який генерується IntelliJ IDEA Ultimate, з тією ж структурою папок і головним класом, що запускає програму, як вказано на рисунку 3.8.

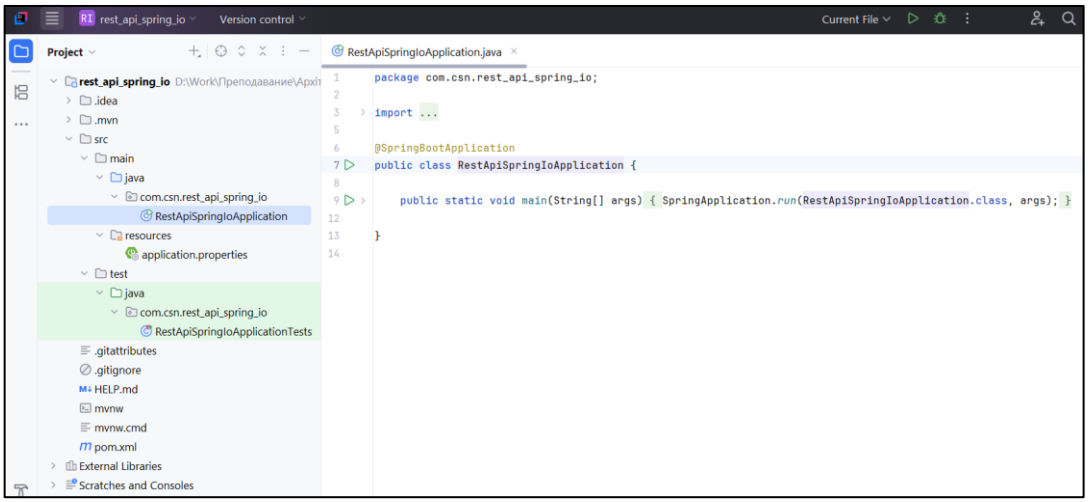

Рисунок 3.8 – Головний клас застосунку

Створений проект має запускатися (якщо запустити клас RestApiSpringIoApplication), як на рисунку 3.9.

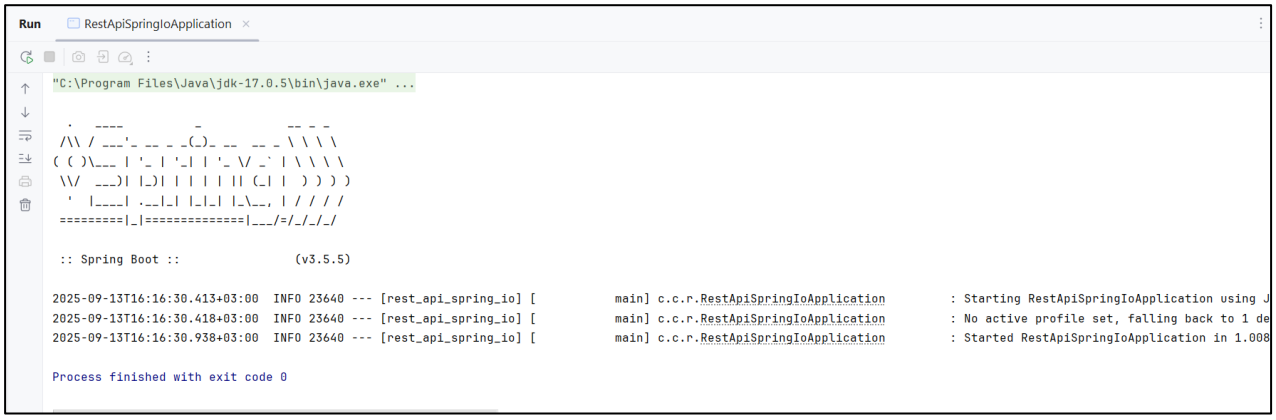

Рисунок 3.9 – Консольний вивід у разі вдалого запуску застосунка

Створення REST функціоналу. Наш додаток працює з клієнтами. Тому насамперед необхідно визначити сутність клієнта у вигляді POJO класу.

Створимо пакет model всередині пакета com.csn.rest_api. Всередині model створимо клас Client, як показано в лістингу 3.5.

Лістинг 3.5. – Код класу Client

```
package com.csn.rest_api.model;

public class Client {

    private Integer id;
    private String name;
    private String email;
    private String phone;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
```

Кінець лістинга 3.5

```
    }
    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }
}
```

У сервісі будуть реалізовані CRUD операції для клієнта. Наступним кроком слід створити сервіс, у якому ці операції будуть реалізовані. У пакеті com.csn.rest_api створимо пакет service, всередині якого визначимо інтерфейс ClientService.

Далі потрібно створити реалізацію цього інтерфейсу. На даному етапі сховищем для клієнтів буде Map<Integer, Client>, де ключем є id клієнта, а значенням – сам клієнт. Це зроблено, щоб не ускладнювати лабораторну роботу деталями роботи з БД. У наступних лабораторних роботах буде реалізовано взаємодію з реальною базою даних.

У пакеті service створимо реалізацію інтерфейсу ClientService, як показано в лістингу 3.6.

Лістинг 3.6. – Код класу ClientServiceImpl

```
package com.csn.rest_api.service;

import com.csn.rest_api.model.Client;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.HashMap;
```

Продовження лістингу 3.6

```
import java.util.List;
import java.util.Map;
import java.util.concurrent.atomic.AtomicInteger;
@Service
public class ClientServiceImpl implements ClientService {

    // Сховище клієнтів
    private static final Map<Integer, Client>
CLIENT_REPOSITORY_MAP = new HashMap<>();

    // Змінна для генерації ID клієнта
    private static final AtomicInteger CLIENT_ID_HOLDER = new
AtomicInteger();

    @Override
    public void create(Client client) {
        final int clientId =
CLIENT_ID_HOLDER.incrementAndGet();
        client.setId(clientId);
        CLIENT_REPOSITORY_MAP.put(clientId, client);
    }

    @Override
    public List<Client> readAll() {
        return new
ArrayList<>(CLIENT_REPOSITORY_MAP.values());
    }
    @Override
    public Client read(int id) {
        return CLIENT_REPOSITORY_MAP.get(id);
    }
```

Кінець лістинга 3.6

```
    @Override
    public boolean update(Client client, int id) {
        if (CLIENT_REPOSITORY_MAP.containsKey(id)) {
            client.setId(id);
            CLIENT_REPOSITORY_MAP.put(id, client);
            return true;
        }

        return false;
    }

    @Override
    public boolean delete(int id) {
        return CLIENT_REPOSITORY_MAP.remove(id) != null;
    }
}
```

Анотація @Service повідомляє Spring, що цей клас є сервісом. Це спеціальний тип класу, де реалізується певна бізнес–логіка застосунку. Завдяки цій анотації Spring надалі надаватиме екземпляр цього класу у потрібних місцях за допомогою Dependency Injection.

Тепер створимо контролер – спеціальний клас, у якому реалізується логіка обробки клієнтських запитів на певному ендпоінті (URI).

Створіть пакет controller всередині пакета com.csn.rest_api. Усередині controller створіть клас ClientController.

Для кращого розуміння створюватимемо цей клас поетапно. Спершу створимо сам клас і впровадимо в нього залежність від ClientService, як показано в лістингу 3.7.

Лістинг 3.7 – Код класу ClientController

```
package com.csn.rest_api.service;

import
org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.web.bind.annotation.RestController;

@RestController
public class ClientController {

    private final ClientService clientService;

    @Autowired
    public ClientController(ClientService clientService) {
        this.clientService = clientService;
    }
}
```

Пояснення анотацій:

@RestController – повідомляє Spring, що цей клас є REST–контролером, тобто в ньому реалізується логіка обробки клієнтських запитів.

@Autowired – повідомляє Spring, що в цьому місці слід впровадити залежність. У конструктор передається інтерфейс ClientService. Реалізацію цього сервісу ми раніше позначили анотацією @Service, і тепер Spring зможе передати екземпляр цієї реалізації у конструктор контролера.

Далі покроково реалізовуватимемо кожен метод контролера для обробки CRUD–операцій. Почнемо з операції Create, для чого напишемо метод create:

```
@PostMapping(value = "/clients")
public ResponseEntity<?> create(@RequestBody Client client) {
  clientService.create(client);
  return new ResponseEntity<>(HttpStatus.CREATED);}
```

Розбір методу:

@PostMapping(value = "/clients") – вказує, що цей метод обробляє POST–запити за адресою /clients.

Метод повертає ResponseEntity<?>. ResponseEntity – це спеціальний клас для формування відповідей, який дозволяє надсилати клієнту HTTP статус–код.

Метод приймає параметр @RequestBody Client client, значення якого підставляється з тіла запиту. Про це повідомляє анотація @RequestBody.

У тілі методу викликаємо метод create сервісу і передаємо йому клієнта, отриманого через параметри контролера.

Після цього повертаємо статус 201 Created, створивши новий об’єкт ResponseEntity та передавши йому потрібне значення енума HttpStatus.

Реалізація операції Read.

Реалізуємо операцію отримання списку всіх наявних клієнтів:

```
@GetMapping(value = "/clients")
public ResponseEntity<List<Client>> read() {
  final List<Client> clients = clientService.readAll();
  return clients != null &&  !clients.isEmpty()
        ? new ResponseEntity<>(clients, HttpStatus.OK)
        : new ResponseEntity<>(HttpStatus.NOT_FOUND);
}
```

Пояснення до створеного методу:

@GetMapping(value = "/clients") – аналогічно до @PostMapping, але тепер метод обробляє GET–запити.

Цього разу метод повертає ResponseEntity<List<Client>>, тобто окрім HTTP статусу, ми також повертаємо тіло відповіді у вигляді списку клієнтів.

У REST–контролерах Spring усі POJO об’єкти та колекції POJO об’єктів, які повертаються як тіло відповіді, автоматично серіалізуються в JSON, якщо явно не вказано інше. Це цілком підходить для наших потреб.

У тілі методу за допомогою сервісу отримуємо список усіх клієнтів. Якщо список не null і не порожній, повертаємо через ResponseEntity сам список клієнтів та HTTP статус 200 OK. Інакше повертаємо лише HTTP статус 404 Not Found.

Реалізація можливості отримувати клієнта за його id: У цьому методі з'являється змінна шляху в URI: value = "/clients/{id}". Значення, що визначене у фігурних дужках, передається в параметри методу як змінна типу int за допомогою анотації @PathVariable(name = "id").

Метод приймає запити на URI виду /clients/{id}, де замість {id} може бути будь–яке числове значення. Це значення передається змінній int id – параметру методу.

У тілі методу за допомогою сервісу отримуємо об'єкт Client за переданим id. Далі, за аналогією з методом для списку, повертаємо або HTTP статус 200 OK і сам об'єкт Client, або лише статус 404 Not Found, якщо клієнт з таким id не знайдений у системі.

Залишилося реалізувати дві операції – Update і Delete. У цих методах нічого суттєво нового немає, тому детальний опис пропускаємо. Варто лише зазначити: метод update обробляє PUT–запити (@PutMapping), а метод delete обробляє DELETE–запити (@DeleteMapping).

Повний код контролера наведено в лістингу 3.8.

Лістинг 3.8 – Повний код класу контролера

```
package com.csn.rest_api.controller;
import com.csn.rest_api.model.Client;
import com.csn.rest_api.service.ClientService;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import java.util.List;
```

Продовження лістингу 3.8

```
@RestController
public class ClientController {
    private final ClientService clientService;
    @Autowired
    public ClientController(ClientService clientService) {
        this.clientService = clientService;
    }
    @PostMapping(value = "/clients")
    public ResponseEntity<?> create(@RequestBody Client
client) {
        clientService.create(client);
        return new ResponseEntity<>(HttpStatus.CREATED);
    }
    @GetMapping(value = "/clients")
    public ResponseEntity<List<Client>> read() {
        final List<Client> clients = clientService.readAll();
        return clients != null &&  !clients.isEmpty()
                ? new ResponseEntity<>(clients,
HttpStatus.OK)
                : new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    @GetMapping(value = "/clients/{id}")
    public ResponseEntity<Client> read(@PathVariable(name =
"id") int id) {
        final Client client = clientService.read(id);
        return client != null
                ? new ResponseEntity<>(client, HttpStatus.OK)
                : new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    @PutMapping(value = "/clients/{id}")
    public ResponseEntity<?> update(@PathVariable(name =
"id") int id, @RequestBody Client client) {
```

Кінець лістинга 3.8

```
        final boolean updated = clientService.update(client,
id);
        return updated
                ? new ResponseEntity<>(HttpStatus.OK)
                : new
ResponseEntity<>(HttpStatus.NOT_MODIFIED);
    }
    @DeleteMapping(value = "/clients/{id}")
    public ResponseEntity<?> delete(@PathVariable(name =
"id") int id) {
        final boolean deleted = clientService.delete(id);
        return deleted
                ? new ResponseEntity<>(HttpStatus.OK)
                : new
ResponseEntity<>(HttpStatus.NOT_MODIFIED);
    }
}
```

У результаті структура нашого проекту виглядає як на рисунку 3.10:

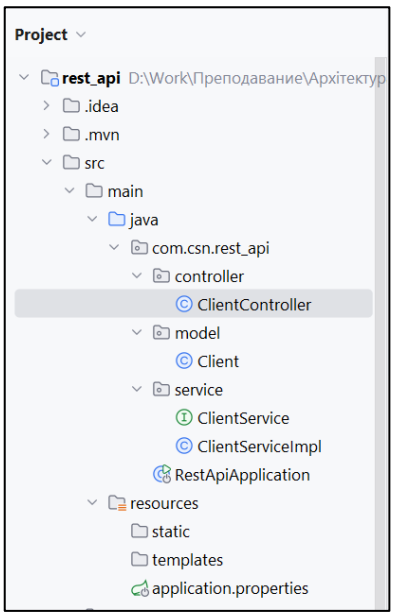

Рисунок 3.10 – Фінальний вигляд структури проекту

3.2.1 Запуск та тестування

Щоб запустити наш застосунок, достатньо запустити метод main у класі RestExampleApplication.

Для тестування знадобиться плагін у браузері або застосунок, які вміють робити прямі HTTP–запити до сервера. Наприклад, можна використовувати плагін RESTClient для Mozilla Firefox або аналогічний за функціональністю для Google Chrome, або завантажити для цього окремий застосунок, наприклад Postman (https://www.postman.com/).

Тестування з використанням застосунку Postman. Головне вікно програми Postman наведено на рисунку 3.11.

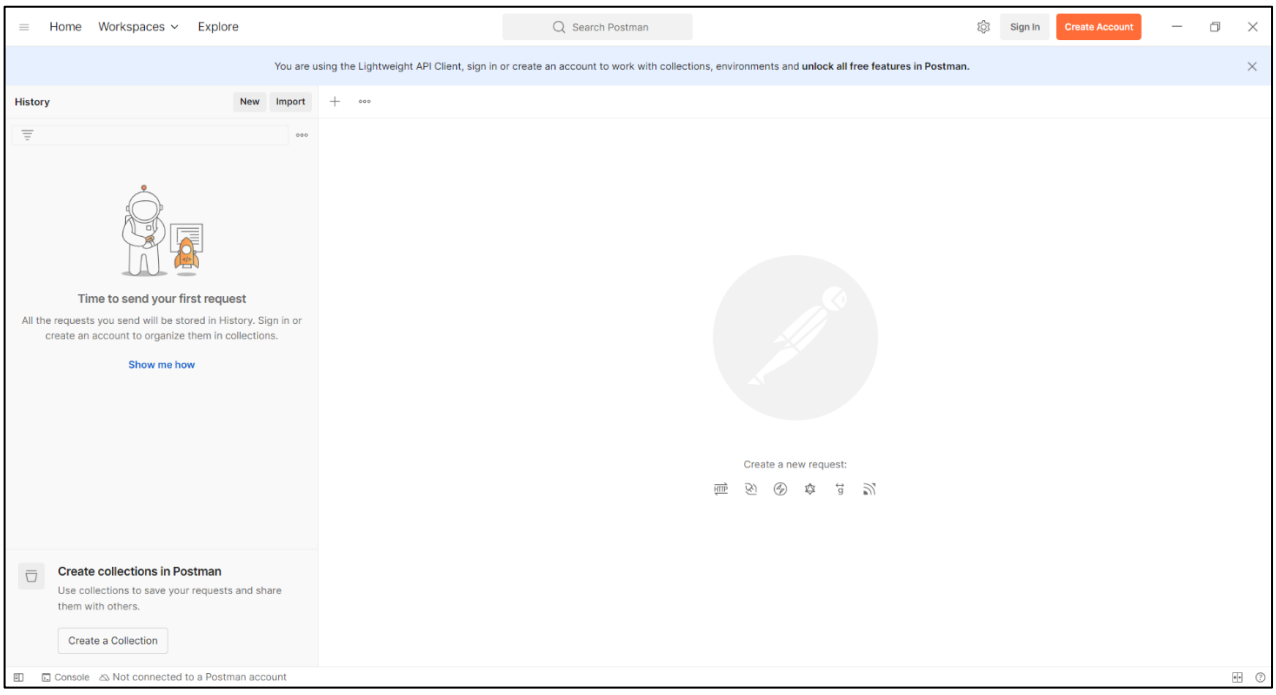

Рисунок 3.11 – Головне вікно програми Postnam

Натискаємо кнопку New у верхньому лівому куті. Вибираємо HTTP, як показано на рисунку 3.12.

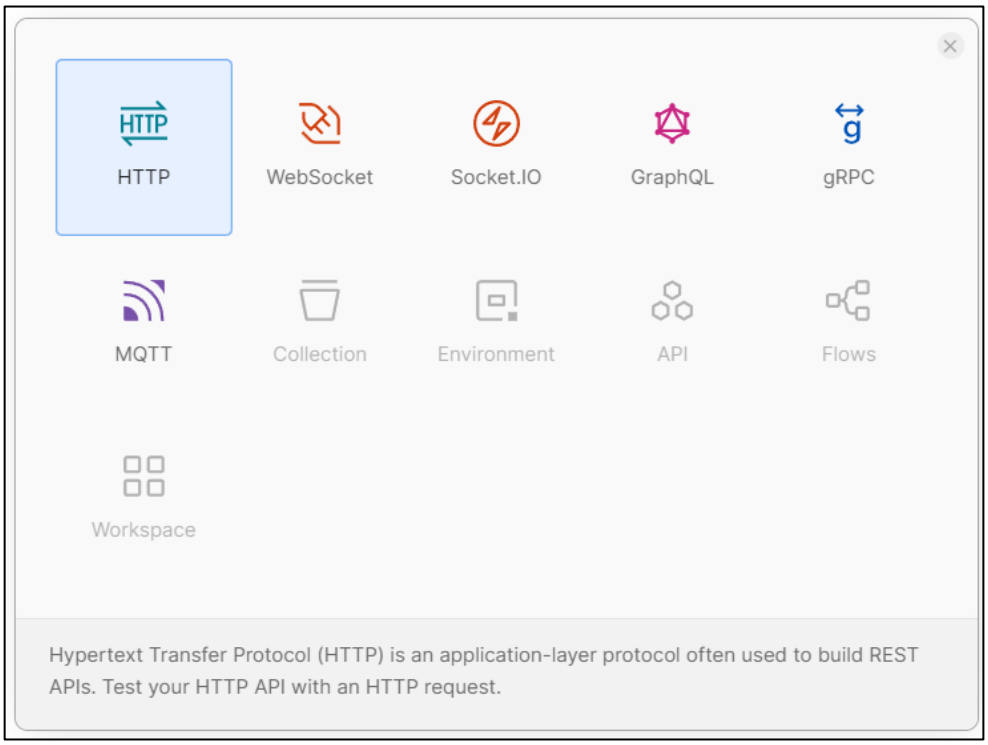

Рисунок 3.12 – Створення нового запиту в програмі Postnam

Тепер надішлемо POST–запит на сервер до ендпоінту http://localhost:8080/clients і створимо першого клієнта (виберіть вкладку Body і тип контенту JSON, як на рисунку 3.13)

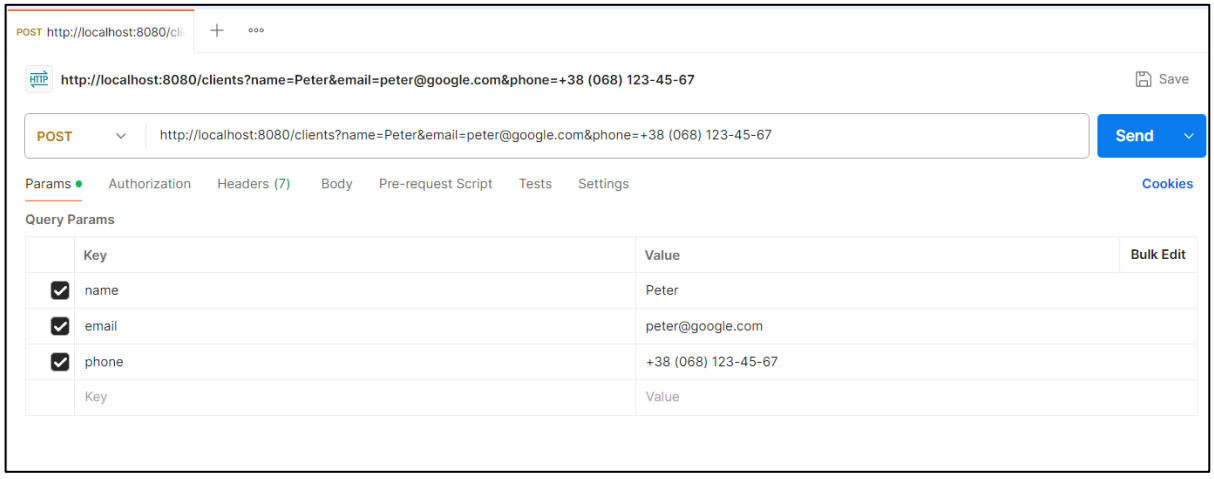

Рисунок 3.13 – Параметри POST запиту до сервера

Введіть текст:

```
{
    "name": "Test",
    "email": "test@gmail.com",
    "phone": "+38 (068) 123-45-67"
}
```

У результаті ви повинні отримати відповідь зі статус–кодом 201 – Created, як на рисунку 3.14.

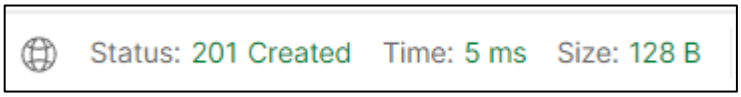

Рисунок 3.14 – Приклад відповіді 201 – Created

Створіть таким чином кілька користувачів.

Для отримання списку користувачів необхідно виконати GET–запит до сервера, вказавши в якості URL ендпоінт http://localhost:8080/clients, як показано на рисунку 3.15.

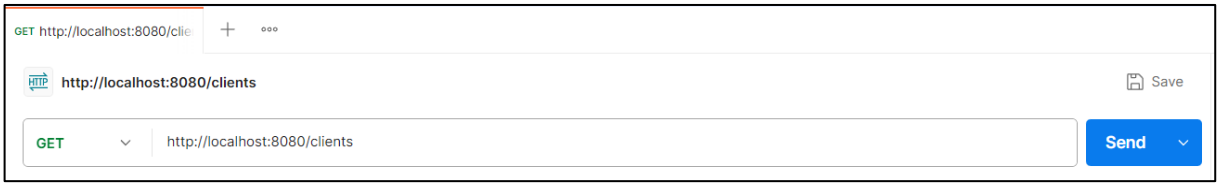

Рисунок 3.15 – Приклад GET запиту

У результаті ми повинні отримати список користувачів, яких щойно створили на сервері, як наведено на рисунку 3.16.

Рисунок 3.16 – Список користувачів у відповіді сервера

Також можна отримати дані одного конкретного користувача, виконавши GET–запит до ендпоінту http://localhost:8080/clients/<id>, наприклад як на рисунках 3.17 та 3.18.

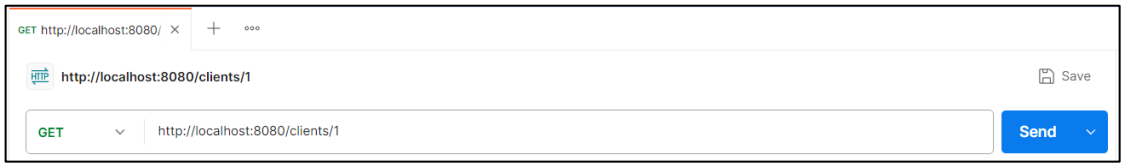

Рисунок 3.17 – Приклад GET запиту до даних конкретного користувача

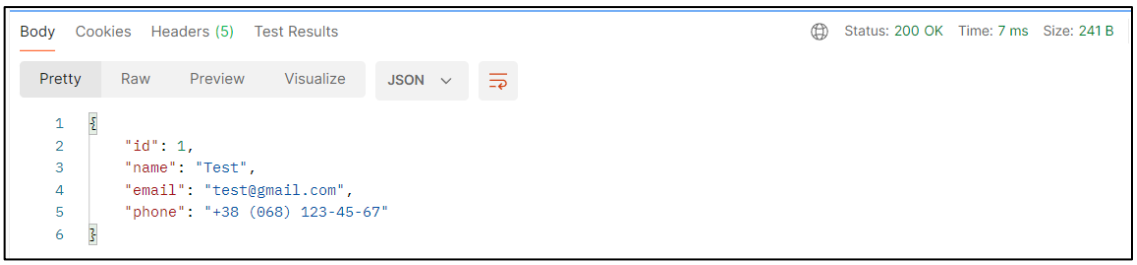

Рисунок 3.18 – Приклад відповіді сервера з даними користувача

3.2.2 Тестування з використанням плагіна REST Client

За аналогією з застосунком Postman можна використовувати плагіни для браузера, які широко представлені для всіх популярних браузерів. Для демонстрації далі буде використаний плагін RESTClient для Mozilla Firefox (http://restclient.net/).

Створення нового запису. Оскільки запит до сервера робиться у форматі JSON, необхідно додати кастомний заголовок до HTTP–запиту. Для цього у верхньому меню оберіть пункт Headers → Custom Header, з'явиться вікно із налаштуваннями заголовків запиту, як на рисунку 3.19.

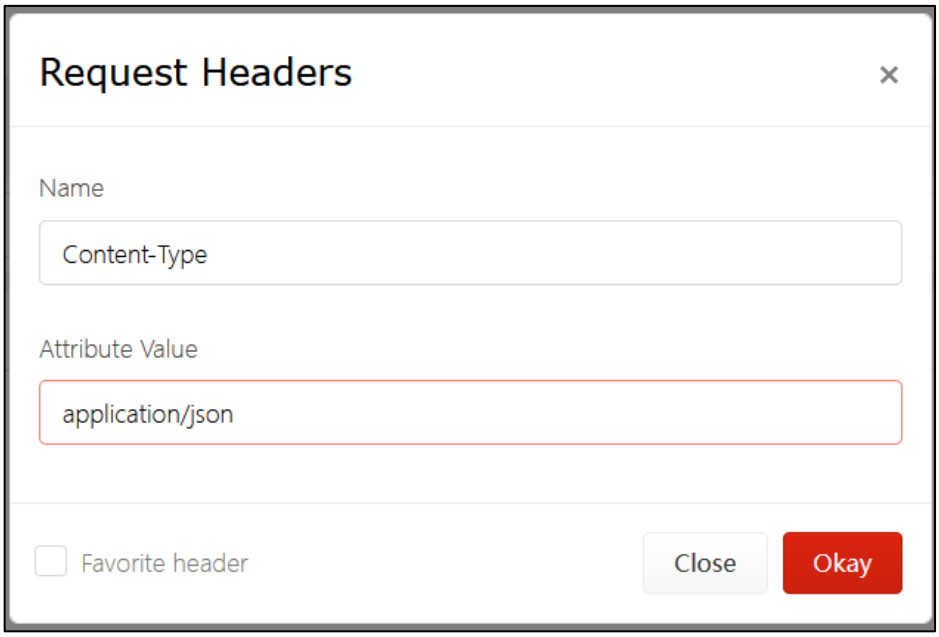

Рисунок 3.19 – Вікно з налаштуваннями заголовків запиту

Після цього можна робити POST–запит, такий же, як ми робили з Postman, як показано на рисунку 3.20.

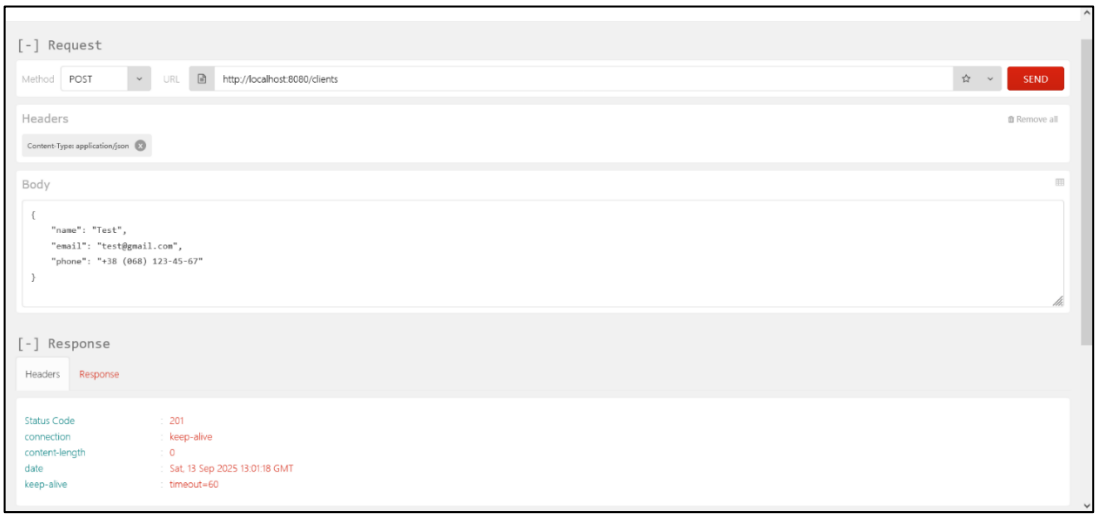

Рисунок 3.20 – Приклад тіла POST запиту із додатковими заголовками

Аналогічно можна запросити список усіх записів, використовуючи метод GET, як показано на рисунку 3.21.

Рисунок 3.21 – Приклад GET запиту до сервера

Приклад відповіді сервера наведен на рисунку 3.22.

Рисунок 3.22 – Приклад відповіді сервера

Перевагою плагіна REST Client є те, що він показує всі виконані запити у вигляді консольних команд для утиліти Curl, приклад команди наведено на рисунку 3.23.

Рисунок 3.23 – Приклад команди curl що відповідає запиту до сервера

3.3 Завдання до лабораторної роботи

Реалізувати REST API для каталогу книг. Каталог повинен містити такі пов’язані сутності:

– автор (author) – з полями: ім’я (name), рік народження (year_of_birth);

– жанр (genre) – з єдиним полем назва (title);

– книга (book) – з полями: назва (title), рік видання (year), автор (author_id), жанр (genre_id).

Сутність «книга» повинна бути пов’язана з сутностями «автор» і «жанр» відповідними полями.

Необхідно реалізувати повний набір CRUD методів для всіх трьох типів об’єктів, а також методи:

– findBooksByAuthor(author_id) – для пошуку всіх книг певного автора;

– findBooksByGenre(genre_id) – для пошуку всіх книг певного жанру;

– findAuthorGenres(author_id) – для пошуку всіх жанрів, у яких писав автор;

– findBooksByYear(year) – для пошуку всіх книг, виданих у певному році;

– findAuthorsByYear(year) – для пошуку всіх авторів, народжених у певному році.

Також необхідно передбачити коректну обробку можливих помилок у параметрах (наприклад, запит неіснуючого автора або створення книги з неіснуючим ідентифікатором автора чи жанру) і повертати відповідні помилки у вигляді HTTP Status Code + опис помилки.

При видаленні записів будь–якої з сутностей необхідно контролювати консистентність даних та або видаляти всі пов’язані сутності (наприклад, при видаленні автора видаляти всі його книги), або повертати повідомлення про помилку (у вигляді відповідного статус–коду) з поясненням, у чому саме проблема та як її виправити.

Список HTTP Status Codes можна переглянути у Вікіпедії (https://en.wikipedia.org/wiki/List_of_HTTP_status_codes) і коректно повертати найбільш підходящий код у кожній ситуації (201 – Created при створенні запису, 200 – OK при запиті існуючого запису, 404 – Not Found при запиті неіснуючого запису тощо).

3.4 Зміст звіту

1. Формулювання мети й задачі лабораторної роботи.
2. Власний програмний код, розроблений під час виконання лабораторної роботи із коментарями
3. Короткі відповіді на контрольні запитання
4. Висновки за результатами роботи.

3.5 Контрольні питання

1. Що таке REST API і які основні принципи архітектури REST слід дотримуватися при розробці сервісу?
2. Які HTTP–методи використовуються для CRUD–операцій у REST API і яка їх відповідність операціям створення, читання, оновлення та видалення?

3. Як у Spring Boot створюється REST–контролер і яку роль виконує анотація @RestController?

4. Для чого використовується анотація @RequestMapping або її спеціалізовані варіанти (@GetMapping, @PostMapping тощо)?

5. Як у Spring Boot реалізується передача параметрів у REST–запитах (через path variable, query parameter або request body)?

6. Як обробляються помилки у REST API та які типові коди HTTP–відповідей слід повертати для CRUD–операцій?

ЛАБОРАТОРНА РОБОТА № 4 – REST API ІЗ ЗАСТОСУВАННЯМ MYSQL

Мета роботи: одержати знання та навички розробки REST API, як обгортки поверх даних, що зберігаються в реляційній базі даних MySQL із використання фреймворку Spring Boot [1–7].

4.1 Теоретичні відомості

4.1.1 Що таке Spring Data JPA

Spring Data JPA – це підпроєкт Spring, який спрощує роботу з базами даних через JPA (Java Persistence API). Він надає високорівневі абстракції над стандартною JPA/Hibernate, зменшує бойлерплейт–код (репозиторії, CRUD–операції, запити) і додає потужні можливості (породження запитів за іменем, пагінація, сортування, специфікації тощо). У поєднанні зі Spring Boot отримуємо майже «з коробки» працюючу шар збереження даних.

Основні ідеї та компоненти Repository abstraction. Інтерфейси типу JpaRepository<T, ID> надають готові методи save, findById, findAll, delete тощо. Не треба писати реалізації для базових операцій.

Derived query methods. Методам інтерфейсу можна давати імена за шаблоном (наприклад findByEmailAndActiveTrue) – Spring Data автоматично згенерує потрібний запит.

@Query**.** Якщо потрібен складніший запит, можна використовувати JPQL або нативний SQL через анотацію @Query.

Paging & Sorting. Вбудована підтримка пагінації (Page, Pageable) і сортування (Sort).

Specifications / Criteria. Інтерфейс JpaSpecificationExecutor дає змогу будувати динамічні запити за допомогою Specifications (Criteria API).

Custom repository implementation. Можна доповнювати репозиторій власними методами, реалізованими через EntityManager.

Аудит і квабіти (Auditing). Підтримка автоматичного заповнення полів createdDate, lastModifiedDate, createdBy тощо.

Інтеграція зі Spring Boot. Достатньо додати spring–boot–starter–data–jpa і налаштувати spring.datasource.* – автоконфігурація підключить EntityManager, TransactionManager тощо.

Entity (Сутність). Entity – це клас Java, який відображає таблицю у базі даних. Кожен об’єкт сутності відповідає одному рядку таблиці. Сутності позначаються анотацією @Entity, а поля – колонками таблиці (за замовчуванням ім’я поля збігається з ім’ям колонки, але можна змінювати через @Column).

– кожна сутність повинна мати первинний ключ (@Id) для унікальної ідентифікації запису;

– значення ключа може генеруватися автоматично (@GeneratedValue);

– сутності можуть бути пов’язані між собою через асоціації (@OneToMany, @ManyToOne, @OneToOne, @ManyToMany);

– сутність керується EntityManager, який відповідає за збереження, оновлення, видалення та підвантаження об’єктів.

Repository (Репозиторій). Репозиторій у JPA – це абстракція для роботи з сутностями, яка приховує деталі EntityManager і SQL–запитів. Він дозволяє виконувати операції CRUD (Create, Read, Update, Delete) та інші запити без написання низькорівневого коду.

У Spring Data JPA репозиторій зазвичай створюється як інтерфейс, що наслідує JpaRepository<Entity, ID>:

– методи save(), findById(), findAll(), delete() доступні «з коробки»;

– можна створювати похідні методи (findByEmail, findByAgeGreaterThan) – Spring Data автоматично згенерує потрібні запити;

– для складних умов можна використовувати @Query (JPQL або SQL).

Service (сервісний шар) У Spring–проєктах шар сервісу (Service layer) відповідає за бізнес–логіку програми та координацію роботи між контролерами і репозиторіями.

Призначення:

– інкапсулює правила бізнес–логіки та перевірки даних;

– виконує складні операції з кількома репозиторіями;

– забезпечує транзакційність операцій (@Transactional);

– відділяє безпосередню роботу з базою (репозиторії) від контролерів, що відповідають за REST API.

Взаємодія з JPA:

– сервіс працює із репозиторіями, які під капотом використовують EntityManager;

– методи сервісу викликають save(), findById(), delete() та інші методи репозиторіїв;

– завдяки анотації @Transactional сервіс гарантує цілісність даних під час виконання кількох операцій.

Приклад найпростіших класів. Клас Entity:

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;
    @Version
    private Long version; // для оптимістичної блокування
}
```

Клас Repository

```
public interface UserRepository extends JpaRepository<User,
Long>, JpaSpecificationExecutor<User> {
    Optional<User> findByEmail(String email);
    List<User> findByNameContainingIgnoreCase(String
fragment);
    @Query("SELECT u FROM User u WHERE u.email
LIKE %:domain")
    List<User> findByEmailDomain(@Param("domain") String
domain);
}
```

Приклад класу сервісу (використання репозиторію)

```
@Service
public class UserService {
private final UserRepository repo;
@Autowired
public UserService(UserRepository repo) { this.repo = repo; }
public User create(User u) { return repo.save(u); }
public Page<User> list(int page, int size) {
    return repo.findAll(PageRequest.of(page, size,
Sort.by("name")));
}
@Transactional
public void delete(Long id) { repo.deleteById(id); }
}
```

4.1.2 Типи репозиторіїв у Spring Data JPA

Spring Data JPA надає кілька інтерфейсів репозиторіїв, які розширюють один одного та надають різний рівень функціональності. Основні типи:

CrudRepository. Найбільш базовий інтерфейс для роботи з сутностями. Підтримує стандартні CRUD–операції**:**

– save(S entity) – створення або оновлення;

– findById(ID id) – отримання за ідентифікатором;

– findAll() – отримання всіх записів;

– deleteById(ID id) – видалення за ID;

– existsById(ID id) – перевірка наявності.

PagingAndSortingRepository Розширює CrudRepository і додає можливість пагінації та сортування.

Додаткові методи:

– findAll(Sort sort) – повертає всі записи з сортуванням;

– findAll(Pageable pageable) – повертає частину даних (сторінку).

JpaRepository. Найбільш повний інтерфейс, який розширює PagingAndSortingRepository.

Додає додаткові методи:

– flush() – примусове скидання змін у базу;

– saveAndFlush() – зберегти і одразу синхронізувати з базою;

– deleteInBatch() / deleteAllInBatch() – видалення групи записів без завантаження в пам’ять.

JpaRepository **–** це найпопулярніший вибір для Spring Boot–проєктів, бо включає всі функції: CRUD, сортування, пагінацію та додаткові методи для керування транзакціями.

4.1.3 Запити в Spring Data JPA

Spring Data JPA надає кілька способів створення запитів до бази даних. Найчастіше використовуються:

1. Похідні запити (Derived Queries)

2. JPQL–запити

3. Нативні SQL–запити

4. Проєкції (Projections)

5. Розглянемо кожен із них більш детально.

Похідні запити (Derived Queries). Це запити, які генеруються автоматично на основі назви методу в інтерфейсі Repository.

Приклад:

```
public interface UserRepository extends JpaRepository<User,
Long> {
    List<User> findByLastName(String lastName);
    User findByEmail(String email);
    List<User> findByAgeGreaterThan(int age);
}
```

З приклада можна побачити, що:

– findByLastName("Petrenko") → знайде всіх користувачів з прізвищем *Petrenko*.

– findByAgeGreaterThan(18) → знайде всіх користувачів старше 18 років.

– Spring Data JPA розбирає назву методу та будує SQL–запит автоматично.

JPQL–запити. JPQL (Java Persistence Query Language) – це мова запитів, схожа на SQL, але працює з сутностями (Entity), а не безпосередньо з таблицями.

Приклад:

```
public interface UserRepository extends JpaRepository<User,
Long> {
@Query("SELECT u FROM User u WHERE u.age > :age")
    List<User> findUsersOlderThan(@Param("age") int age);
}
```

Тут використовується сутність User, а не назва таблиці users. JPQL–запити перетворюються на SQL під час виконання.

Нативні запити (Native Queries). У деяких випадках потрібен повний контроль над SQL–запитом (наприклад, специфічні функції СУБД). Тоді можна використовувати нативні SQL–запити**.**

Приклад:

```
public interface UserRepository extends JpaRepository<User,
Long> {
@Query(value = "SELECT * FROM users u WHERE u.email
= :email", nativeQuery = true)
User findByEmailNative(@Param("email") String email);
}
```

З приклада можна побачити, що nativeQuery = true означає, що запит буде виконано як «чистий SQL». Крім того у цьому запиті використовуються назви таблиць і колонок з бази даних.

Проєкції (Projections). Часто немає потреби отримувати всю сутність, а достатньо лише кількох полів. Для цього використовуються проєкції**.**

Є два типи проєкцій:

– інтерфейсні проєкції;

– класові (DTO) проєкції.

Інтерфейсна проєкція. Клас представлення:

```
public interface UserView {
    String getFirstName();
    String getLastName();
}
```

Репозиторій, що вертає список об’єктів типу UserView (перетворення відбувається автоматично):

```
public interface UserRepository extends JpaRepository<User,
Long> {
    List<UserView> findByAgeGreaterThan(int age);
}
```

Виконуючи findByAgeGreaterThan(18), ми отримаємо лише firstName та lastName, без інших полів.

4.1.4 DTO–проєкція (через конструктор)

DTO (Data Transfer Object) – це спеціальний об’єкт, призначений для передачі даних між шарами програми, наприклад, між сервером і клієнтом у REST API. DTO використовується, щоб обмежити передавані поля лише необхідними даними, приховати внутрішню структуру сутності, зменшити об’єм трафіку та підвищити безпеку, а також спростити серіалізацію у JSON або інші формати.

Клас DTO об’єкта:

```
public class UserDTO {
private final String firstName;
private final String email;
public UserDTO(String firstName, String email) {
    this.firstName = firstName;
    this.email = email;
}
// геттери
}
```

Клас репозиторія, що відразу повертає DTO об’єект:

```
public interface UserRepository extends JpaRepository<User,
Long> {
@Query("SELECT new com.example.demo.dto.UserDTO(u.firstName,
u.email) FROM User u")
    List<UserDTO> findAllUserDTOs();
}
```

Генерація запитів на основі імен методів репозиторію у Spring Data JPA. Однією з найзручніших функцій Spring Data JPA є автоматичне створення SQL/JPQL–запитів на основі назви методу в репозиторії. Це дозволяє отримувати дані без написання власних JPQL чи SQL–запитів.

Основна ідея. Spring Data аналізує назву методу і перетворює її на запит.

Метод починається зі стандартного префіксу:

– findBy – знайти запис(и) за умовою;

– getBy – аналог findBy;

– readBy – також аналог;

– countBy – підрахувати записи, що відповідають умовам;

– existsBy – перевірити, чи існують записи.

Далі вказуються умови через назви полів сутності та ключові слова:

– And, Or, Between, LessThan, GreaterThan, Like, In, IsNull, NotNull, OrderBy.

Приклади

1. Пошук за одним полем:

```
List<User> findByLastName(String lastName);
```

2. Пошук за кількома умовами:

```
List<User> findByFirstNameAndAge(String firstName, int age);
```

Автоматично генерує та виконує JPQL: SELECT u FROM User u WHERE u.firstName = ?1 AND u.age = ?2

3. Використання порівнянь і діапазонів:

```
List<User> findByAgeGreaterThan(int age);
List<User> findByAgeBetween(int start, int end);
```

Автоматично генерує запити:

- GreaterThan → age > ?1
- Between → age BETWEEN ?1 AND ?2

4. Пошук по частковому збігу (LIKE):

```
List<User> findByEmailContaining(String fragment);
```

Автоматично генерує JPQL з LIKE %fragment%

5. Сортування:

```
List<User> findByActiveTrueOrderByLastNameAsc();
```

Знаходить активних користувачів і сортує за прізвищем по зростанню

6. Перевірка існування запису:

```
boolean existsByEmail(String email);
```

Генерує запит, що повертає true, якщо такий email існує

7. Підрахунок записів:

```
long countByAgeGreaterThan(int age);
```

Повертає кількість користувачів старше певного віку

4.1.5 Пагінація та сортування у Spring Data JPA

У більшості застосунків база даних може містити сотні або тисячі записів. Щоб не завантажувати всі дані одразу, використовується пагінація (pagination) – поділ даних на сторінки, а сортування (sorting) дозволяє отримувати записи у потрібному порядку. Spring Data JPA має вбудовану підтримку цих функцій через інтерфейси PagingAndSortingRepository або JpaRepository.

Сортування (Sorting):

– сортування здійснюється за допомогою об’єкта Sort;

– можна вказати напрямок: ASC (за зростанням) або DESC (за спаданням).

Приклад:

```
List<User> users =
userRepository.findAll(Sort.by(Sort.Direction.ASC,
"lastName"));
```

```
List<User> users =
userRepository.findAll(Sort.by("lastName").ascending().and(So
rt.by("age").descending()));
```

У першому пракладі отримаємо список всіх користувачів, відсортований за прізвищем за зростанням. У другому – сортуємо спочатку за прізвищем (ASC), потім за віком (DESC).

Пагінація (Pagination):

– використовується об’єкт Pageable, який визначає номер сторінки, розмір сторінки та опціонально сортування;

– метод репозиторію повертає Page<T> – об’єкт, що містить дані сторінки та додаткову інформацію (загальна кількість сторінок, елементів, чи є наступна/попередня сторінка).

Приклад:

```
Pageable pageable = PageRequest.of(0, 10,
Sort.by("lastName").ascending());
Page<User> page = userRepository.findAll(pageable);
List<User> users = page.getContent(); // записи поточної
сторінки
int totalPages = page.getTotalPages(); // загальна кількість
сторінок
long totalElements = page.getTotalElements(); // загальна
кількість записів
```

Тут 0 – номер сторінки (перша сторінка), 10 – розмір сторінки (кількість записів на сторінку). Пагінація без сортування:

```
Page<User> page = userRepository.findAll(PageRequest.of(1,
20));
```

У цьому випадку, поверне другу сторінку (page = 1) з 20 записами. Використання пагінації з кастомними запитами.

```
Page<User> findByAgeGreaterThan(int age, Pageable pageable);
```

Метод поверне користувачів старших за певний вік, розбитих на сторінки та за потреби відсортованих. Приклад виклику:

```
Page<User> page = userRepository.findByAgeGreaterThan(18,
PageRequest.of(0, 5, Sort.by("firstName").descending()));
```

Поверне першу сторінку, 5 користувачів на сторінку, сортування за ім’ям у зворотному порядку.

4.2 Хід роботи

Завантажте та встановіть MySQL із сайту https://dev.mysql.com/downloads/installer/ Як графічний інтерфейс можна використовувати будь–який зручний MySQL Client, наприклад MySQL Workbench, який можна завантажити із сайту https://dev.mysql.com/downloads/workbench/ Зовнішній вигляд інтерфейсу MySQL Workbench показано на рисунку 4.1.

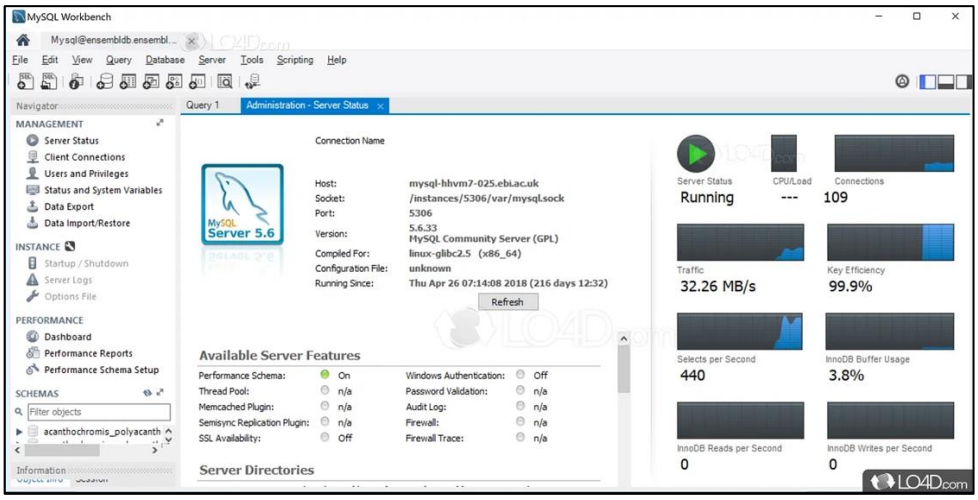

Рисунок 4.1 – Головне вікно програми MySQL Workbench

Альтернативно можна використовувати вбудований у IntelliJ IDEA плагін для роботи з БД. Для цього необхідно встановити безкоштовний плагін Database Navigator.

Відкрийте меню File → Settings → Plugins і у вікні, що з’явиться, у верхньому полі пошуку введіть фразу: database.

У списку результатів пошуку виберіть плагін Database Navigator і натисніть Install, як показано на рисунку 4.2.

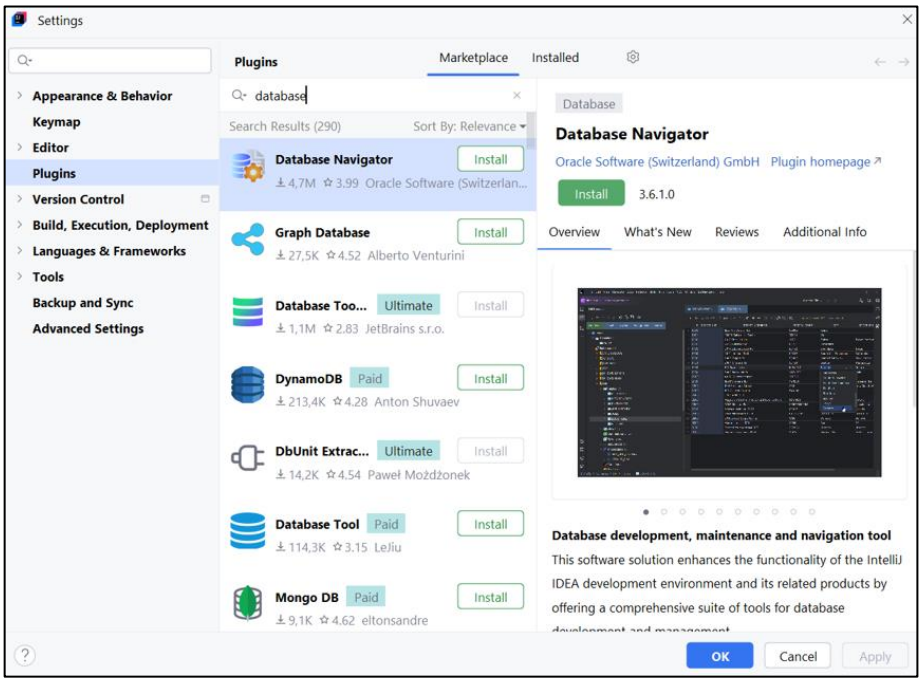

Рисунок 4.2 – Встановлення плагіну Database Navigator

Створіть у MySQL базу даних для лабораторної роботи. Також створіть користувача і надайте йому права на створену базу даних.

У прикладі база даних називатиметься atws, ім’я користувача – atws, а пароль – 123456. Ви можете вказати власні значення.

Налаштування Spring Boot для роботи з MySQL. Для роботи з MySQL необхідно у файлі pom.xml у корені проєкту в розділ <dependencies> додати залежності:

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>mysql</groupId>
```

```
<artifactId>mysql-connector-java</artifactId>
<version>8.0.33</version>
</dependency>
```

Параметри підключення до БД вказуються у файлі /src/resources/application.properties. Для бази даних і користувача, створених раніше, додайте такі значення:

```
spring.datasource.url=jdbc:mysql://localhost:3306/atws
spring.datasource.username=atws
spring.datasource.password=123456
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
spring.jpa.database=MYSQL
```

Запустіть застосунок і переконайтеся (за виводом у консолі під час запуску), що підключення до БД відбулося без помилок. Ознакою успішного підключення до бази даних будуть рядки в консолі на кшталт:

Database info:
Database JDBC URL [Connecting through datasource ‘HikariDataSource (HikariPool–1)’]
Database version: 5.5.25

Створення таблиці в базі даних та ініціалізація її даними. У базі даних необхідно створити таблицю clients із такою структурою:

```
CREATE TABLE IF NOT EXISTS `clients` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`name` varchar(255) NOT NULL,
`email` varchar(255) NOT NULL,
```

```
`phone` varchar(20) NOT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1 ;
```

І заповнити її прикладами значень:

```
INSERT INTO `clients` (`id`, `name`, `email`, `phone`) VALUES
(1, 'Mike', 'mike@gmail.com', '+38 (068) 123-56-67'),
(2, 'James', 'james@google.com', '+38 (068) 234-56-78');
```

Це можна зробити безпосередньо у MySQL клієнті, або налаштувати Spring Boot так, щоб він під час запуску автоматично створював таблицю в БД та заповнював її даними.

Для цього потрібно:

1. У папці src/main/resources/ створити папку db.

2. У папці db створити файл schema.sql і скопіювати в нього SQL–запит для створення таблиці з прикладу вище.

3. У тій самій папці db створити файл data.sql і скопіювати в нього SQL–запити для ініціалізації таблиці clients з прикладу вище.

4. У файл application.properties потрібно додати (в кінець):

```
spring.sql.init.mode=always
spring.sql.init.encoding=UTF-8
spring.sql.init.schema-locations=classpath:db/schema.sql
spring.sql.init.data-locations=classpath:db/data.sql
# spring.jpa.hibernate.ddl-auto=create
# spring.jpa.defer-datasource-initialization=true
# JPA / debug
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Після цього можна запустити застосунок. Якщо все налаштовано правильно, у базі даних буде створено таблицю clients і заповнено її даними з файлу data.sql.

Структура папок має виглядати як на рисунку 4.3.

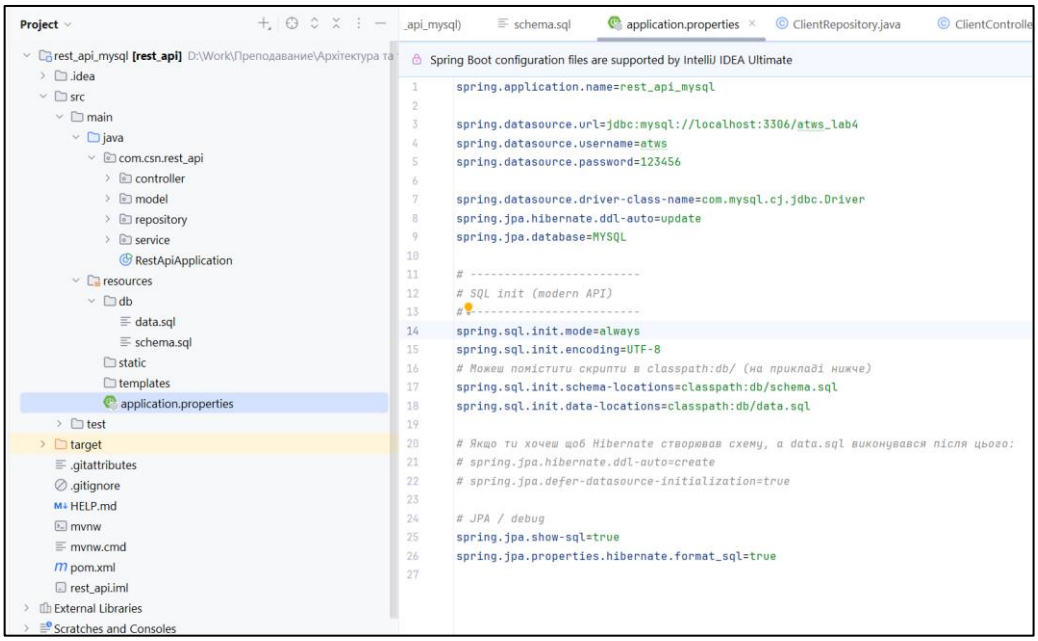

Рисунок 4.3 – Структура папко створеного проекту

Якщо запустити застосунок повторно, буде показано помилку:

```
...
Caused by: java.sql.SQLIntegrityConstraintViolationException:
Duplicate entry '1' for key 'PRIMARY'
...
```

Помилка викликана тим, що дані в БД уже заповнені, і при повторному виконанні запиту з файлу data.sql база даних повертає помилку, оскільки записи з таким первинним ключем уже містяться в БД.

Щоб її прибрати, достатньо закоментувати у файлі application.properties параметр spring.sql.init.data–locations.

У принципі, така ж проблема мала б виникати і при повторному виконанні запиту, що створює таблицю в БД. Помилка не виникає, тому що в самому запиті є доповнення CREATE TABLE IF NOT EXISTS – тобто таблиця створюється

лише, якщо вона відсутня в БД (але сам запит із файлу schema.sql також виконується при кожному запуску програми).

Доопрацювання коду для роботи з БД. Створіть модуль repository і в ньому клас ClientRepository з таким вмістом:

```
package com.csn.rest_api.repository;

import com.csn.rest_api.model.Client;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ClientRepository extends
JpaRepository<Client, Integer> {
}
```

Цей інтерфейс і буде «чарівним чином» взаємодіяти з нашими базами даних і таблицями. Чому чарівним чином? Тому що його реалізацію нам писати не потрібно – її надасть нам фреймворк Spring. Достатньо лише створити такий інтерфейс, і вже можна користуватися цією «магією».

Наступним кроком відредагуємо клас Client, як показано у лістингу 4.1.

Лістинг 4.1 – Код класу Client

```
@Entity
@Table(name = "clients")
public class Client {
    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column(name = "name")
    private String name;
    @Column(name = "email")
```

Кінець лістинга 4.1

```
    private String email;
    @Column(name = "phone")
    private String phone;
//… getters and setters
}
```

У цьому класі ми лише додали деякі анотації:

@Entity – вказує, що цей бін (клас) є сутністю.

@Table – задає ім’я таблиці, яка відображатиметься через цю сутність.

@Id – позначає колонку як первинний ключ (значення, яке забезпечує унікальність даних у таблиці).

@Column – задає ім’я колонки, яка відображається у властивість сутності.

@GeneratedValue – вказує, що значення цієї властивості створюватиметься згідно з обраною стратегією.

Імена полів таблиці не обов’язково повинні збігатися з іменами змінних у класі. Наприклад, змінна firstName у класі може відповідати полю таблиці first_name.

Анотації можна розміщувати як безпосередньо біля полів, так і біля їхніх геттерів. Важливо обрати один із способів і дотримуватися його послідовно в усьому проекті.

Тепер перейдемо до класу ClientServiceImpl і перепишемо його відповідно до лістингу 4.2.

Лістинг 4.2 – Код класу ClientRepository

```
@Service
public class ClientServiceImpl implements ClientService {
    @Autowired
    private ClientRepository clientRepository;
    @Override
    public void create(Client client) {
            clientRepository.save(client);
```

Кінець лістинга 4.2

```
    }
    @Override
    public List<Client> readAll() {
        return clientRepository.findAll();
    }
    @Override
    public Client read(int id) {
        return clientRepository.getOne(id);
    }
    @Override
    public boolean update(Client client, int id) {
        if (clientRepository.existsById(id)) {
            client.setId(id);
            clientRepository.save(client);
            return true;
        }
        return false;
    }
    @Override
    public boolean delete(int id) {
        if (clientRepository.existsById(id)) {
            clientRepository.deleteById(id);
            return true;
        }
        return false;
    }
}
```

Як видно з листингу, усе, що ми зробили, – це видалили вже непотрібні нам рядки:

```
private static final Map<Integer, Client>
CLIENT_REPOSITORY_MAP = new HashMap<>();

// Змінна для генерації ID клієнта
private static final AtomicInteger CLIENT_ID_HOLDER = new
AtomicInteger();
```

Замість них ми оголосили наш інтерфейс ClientRepository, а також розмістили над ним анотацію @Autowired, щоб Spring автоматично додав цю залежність у наш клас.

А також делегували всю роботу цьому інтерфейсу, а точніше – його реалізації, яку додасть Spring.

Переходимо до завершального й найцікавішого етапу – тестування нашого застосунку. Відкриваємо програму Postman або плагін до браузера й надсилаємо GET–запит за цією адресою: http://localhost:8080/clients

У результаті отримаємо відповідь сервера:

```
[{
"id": 1,
"name": "Mike",
"email": "mike@gmail.com",
"phone": "+38 (068) 123-56-67"
}, {
"id": 2,
"name": "James",
"email": "james@google.com",
"phone": "+38 (068) 234-56-78"
}]
```

Надсилаємо POST–запит до ендпоінту http://localhost:8080/clients із вмістом:

```
{
"name" : "John",
"email" : "john@google.com",
"phone" : "+38 (068) 345-67-89"
}
```

Маємо отримати відповідь: Status Code: 201

4.2.1 Пошук користувачів за іменем.

У цьому розділі ми додамо можливість пошуку клієнтів за іменем (припустимо, що імена клієнтів можуть повторюватися). Для цього потрібно внести зміни по всьому ланцюжку: ClientRepository → ClientService → ClientServiceImpl → ClientController.

Додаємо метод у репозиторій ClientRepository, який повертатиме всіх користувачів із певним іменем. Для його реалізації використовуємо анотацію @Query. Додайте такий код в інтерфейс ClientRepository (у сучасних версіях Java інтерфейси також можуть містити реалізацію методів):

```
@Query(value = "SELECT * FROM clients WHERE name = :name",
nativeQuery = true)
Optional<Client> findByName(@Param("name") String name);
```

Тепер необхідно додати новий метод в інтерфейс ClientService. Для цього допишіть у кінець класу сигнатуру методу:

```
/**
* Повертає список клієнтів за їхнім іменем
* @param name – ім’я клієнта
* @return – список клієнтів
*/
List<Client> findByName(String name);
```

І його реалізацію в класі ClientServiceImpl:

```
@Override
public List<Client> findByName(String name) {
    return clientRepository.findByName(name)
            .map(List::of) // якщо є клієнт → обгорнемо в
List<Client>
            .orElseGet(List::of); // якщо немає → повернемо
порожній список
}
```

Тепер залишилося додати в контролері ClientController мапінг для отримання списку клієнтів за іменем:

```
@GetMapping("/clients/names/{name}")
public List<Client> getClientsByName(@PathVariable String
name) {
    return clientService.findByName(name);
}
```

Перезапустіть проєкт. Тепер можна запитати список клієнтів за іменем, виконавши GET–запит до URL виду http://localhost:8080/clients/names/, наприклад як на рисунку 4.4.

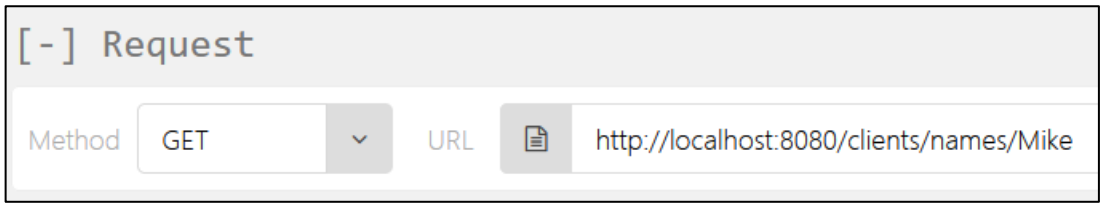

Рисунок 4.4 – Приклад GET запиту до сервера

Поверне результат:

```
[{"id":1,"name":"Mike","email":"mike@gmail.com","phone":"+38
(068) 123-56-67"}]
```

4.3 Завдання до лабораторної роботи

Переробіть завдання до попередньої лабораторної роботи так, щоб каталог книг зберігався в базі даних. На даному етапі достатньо реалізувати 3 незалежні між собою класи моделей і репозиторії для трьох сутностей доменної області: авторів, жанрів і книг. Вся бізнес–логіка щодо створення/читання/зміни/видалення даних у цих пов’язаних таблицях, включно з перевіркою цілісності даних, може бути реалізована в контролері. Для find–методів пошуку даних можна використовувати бізнес–логіку, написану в контролері, або створити в рамках відповідного репозиторію методи з використанням JPA Query Language (включно з параметром nativeQuery = true для зручності роботи з чистим SQL).

Необхідно передбачити контроль відсутності дублікатів жанрів, авторів (за іменем) і книг (за назвою, роком і автором).

Допрацюйте методи отримання списку книг (findBooksByAuthor, findBooksByGenre, findBooksByYear) функціоналом пагінації та сортуванням за автором, жанром і роком відповідно.

Також необхідно передбачити коректну обробку можливих помилок у параметрах (наприклад, запит неіснуючого автора або створення книги з

неіснуючим ідентифікатором автора чи жанру) та виводити відповідні помилки у вигляді HTTP Status Code + опис помилки.

При видаленні записів будь–якої з сутностей потрібно контролювати консистентність даних і або видаляти всі пов’язані сутності (наприклад, при видаленні автора видаляти всі його книги), або повертати повідомлення про помилку (у вигляді відповідного статус–коду) з поясненням, у чому саме вона полягає та як її виправити.

4.4 Зміст звіту

1. Формулювання мети й задачі лабораторної роботи.
2. Власний програмний код, розроблений під час виконання лабораторної роботи із коментарями
3. Короткі відповіді на контрольні запитання
4. Висновки за результатами роботи.

4.5 Контрольні питання

1. Як у Spring Boot налаштовується підключення до бази даних MySQL через application.properties або application.yml?
2. Що таке Entity у JPA і як вона пов’язана з таблицею у базі даних?
3. Яку роль виконують репозиторії (CrudRepository, JpaRepository) у роботі REST API з MySQL?
4. Як реалізується створення, читання, оновлення та видалення даних через REST API із використанням JPA–репозиторіїв?
5. Що таке DTO (Data Transfer Object) і як його застосовують для передачі даних між контролером і клієнтом?
6. Як у Spring Boot реалізується обробка зв’язаних таблиць (наприклад, One–to–Many) через REST API?

7. Які типові коди HTTP–відповідей слід повертати при CRUD–операціях та як їх встановлювати у контролерах?

8. Як у Spring Boot реалізується пагінація та сортування даних у REST API при запитах до бази MySQL?

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Pro RESTful APIs with Micronaut: Build Java–Based Microservices with REST, JSON, and XML / Sanjay Patni // Apress; 3rd edition. 2025 – 187 p.

2. Modern API Development with Spring 6 and Spring Boot 3: Design scalable, viable, and reactive APIs with REST, gRPC, and GraphQL using Java 17 and Spring Boot 3 (2nd ed.). / Sharma, S. // Packt Publishing, 2003 – 494 p.

3. RESTful Web API Patterns and Practices Cookbook: Connecting and Orchestrating Microservices and Distributed Data. / Amundsen, M. // O'Reilly, 2022 – 468 p.

4. REST API Simplified: Developing REST APIs in Spring Boot. / Vijay, S. R. J. // Self–published/independent, 2024 – 155 p.

5. Microservices with Spring Boot 3 and Spring Cloud: Build resilient and scalable microservices using Spring Cloud, Istio, and Kubernetes / Magnus Larsson // Packt Publishing, 3rd edition. 2023. – 706 p.

6. Newman S. Building Microservices: Designing Fine–Grained Systems. 2nd ed. Beijing ; Boston : O'Reilly Media, 2021. 420 p.

7. Spring Microservices in Action / John Carnell, Illary Huaylupo Sánchez // Manning, 2nd edition. 2021 – 448 p.