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 ІІІ

Методичні вказівки до виконання лабораторних робіт з дисципліни «Архітектура і технології вебсервісів» для магістрів спеціальності F7 Комп’ютерна інженерія всіх форм навчання. Частина ІІІ / Укл. М. Б. Ільяшенко – Запоріжжя: НУ «Запорізька політехніка», 2025. – 62с.

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

С.

ВСТУП

Третя частина методичних вказівок з дисципліни «Архітектура і технології вебсервісів» присвячена реалізації складніших сценаріїв роботи REST API у поєднанні зі зв’язаними таблицями бази даних MySQL, а також практиці юніт- та інтеграційного тестування мікросервісів. Якщо у попередній частині студенти ознайомилися з базовими принципами REST і роботою з простою моделлю даних, то тепер акцент зроблено на більш реалістичних і наближених до практики задачах, які виникають під час розробки сучасних корпоративних застосунків.

У рамках лабораторних робіт студенти навчаться працювати з асоціаціями між сутностями в JPA: @OneToOne, @OneToMany, @ManyToOne, @ManyToMany, застосовувати механізми завантаження (fetching), каскадних операцій та обробки видалення й оновлення зв’язаних даних. Це дозволить зрозуміти, як правильно будувати структуру бази даних і відповідні об’єктні моделі, а також як організовувати REST-контролери для взаємодії з такими даними.

Другою важливою темою цієї частини є тестування. Студенти ознайомляться з основними інструментами JUnit, Mockito та MockMvc, а також з підходами до побудови юніт-тестів для перевірки бізнес-логіки і інтеграційних тестів для перевірки роботи всього сервісу разом із базою даних та вебрівнем. Практика написання тестів дозволить сформувати професійний підхід до розробки, коли кожна зміна в коді може бути перевірена автоматизованими засобами.

У результаті виконання цієї частини студенти здобудуть знання та навички, необхідні для створення повноцінних мікросервісних застосунків, що працюють з комплексними моделями даних і відповідають вимогам надійності завдяки якісному тестуванню. Це стане завершальним етапом у формуванні цілісного уявлення про архітектуру та технології вебсервісів, що охоплює як класичні SOAP-рішення, так і сучасні REST API.

ЛАБОРАТОРНА РОБОТА № 5 – REST API ДЛЯ РОБОТИ ЗІ ЗВ'ЯЗАНИМИ ТАБЛИЦЯМИ В MYSQL

Мета роботи: одержати знання та навички створення REST API для роботи зі зв'язаними таблицями в базі даних на прикладі MySQL із використанням сутності доменної моделі у фреймворку Spring Boot [1-6].

1.1 Теоретичні відомості

У JPA зв'язки між сутностями відображаються за допомогою спеціальних анотацій: `@OneToMany`, `@ManyToOne`, `@ManyToMany`, `@OneToOne`. Вони визначають, як об'єкти взаємодіють між собою і як JPA будує SQL-запити та таблиці у базі даних.

1.1.1 Анотація @OneToMany

Відображає зв'язок «один до багатьох», коли один об'єкт є власником колекції інших об'єктів.

Властивості:

– `mappedBy` – ім'я поля у «багато»-сутності, яке є власником зв'язку;

– `cascade` – визначає каскадні операції (`ALL`, `PERSIST`, `MERGE`, `REMOVE`, `REFRESH`, `DETACH`);

– `fetch` – спосіб підвантаження: `LAZY` (відкладене) або `EAGER` (одразу);

– `orphanRemoval` – автоматичне видалення об'єктів, що більше не належать колекції.

Два класи пов'язані відношеннями `@OneToMany` та `@ManyToOne` (ліст. 1.1):

– `Order.user` – власник зв'язку;

– `User.orders` – відображення колекції;

– `mappedBy = "user"` показує, що `orders` не створює додаткову колонку, а використовує `user_id` у таблиці `orders`.

Лістинг 1.1 – Класи пов’язані відношеннями `@OneToMany` та `@ManyToOne`

```
@Entity
public class User {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL,
orphanRemoval = true)
    private List<Order> orders = new ArrayList<>();
}
@Entity
public class Order {
    @Id
    @GeneratedValue
    private Long id;
    private String product;
    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;
}
```

1.1.2 Анотація @ManyToOne

Відображає зв’язок «багато до одного». Клас, що містить `ManyToOne`, зазвичай має зовнішній ключ (`@JoinColumn`) у базі.

Параметри `@JoinColumn`:

– `name` – ім’я колонки у таблиці, що зберігає зовнішній ключ;

– `referencedColumnName` – колонка у зв’язаній таблиці, на яку посилається зовнішній ключ (за замовчуванням – первинний ключ);

– `nullable` – чи може бути `null`;

– `unique` – унікальність значення зовнішнього ключа.

Приклад (див. ліст. 1.1):

```
@ManyToOne
// зовнішній ключ до таблиці User
@JoinColumn(name = "user_id")
private User user;
```

1.1.3 Анотація @ManyToMany

Відображає зв'язок «багато до багатьох», коли об'єкти з обох сторін можуть мати багато зв'язаних елементів. Реалізується через проміжну таблицю (`@JoinTable`).

Параметри `@JoinTable`:

– `name` – ім'я проміжної таблиці;

– `joinColumns` – колонка(-и) для поточної сутності (`@JoinColumn`);

– `inverseJoinColumns` – колонка(-и) для іншої сутності (`@JoinColumn`);

– `uniqueConstraints` – унікальні обмеження на таблиці.

Два класи пов'язані відношеннями `@ManyToMany` (ліст. 1.2):

– таблиця `student_course` зберігає зв'язки між студентами і курсами;

– `mappedBy` у `Course` показує, що власником зв'язку є `Student`.

Лістинг 1.2 – Класи пов'язані відношеннями `@ManyToMany`

```
@Entity
public class Student {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
    @ManyToMany
    @JoinTable(
        name = "student_course",
```

Кінець лістингу 1.2

```
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private Set<Course> courses = new HashSet<>();
}
@Entity
public class Course {
    @Id
    @GeneratedValue
    private Long id;
    private String title;
    @ManyToMany(mappedBy = "courses")
    private Set<Student> students = new HashSet<>();
}
```

1.1.4 Анотація @OneToOne

Відображає зв’язок «один до одного», коли один об’єкт пов’язаний лише з одним іншим об’єктом. Може бути власником або зворотним кінцем зв’язку. Часто використовується з `@JoinColumn` для зовнішнього ключа.

Два класи пов’язані відношеннями `@OneToOne` (ліст. 1.3):

– таблиця `User` має колонку `passport_id`, яка посилається на `Passport.id`;

– `Cascade.ALL` – при збереженні або видаленні користувача, паспорт теж зберігатиметься/видалятиметься.

Лістинг 1.3 – Класи пов’язані відношеннями `@OneToOne`

```
@Entity
public class User {
    @Id
```

Кінець лістингу 1.3

```
    @GeneratedValue
    private Long id;
    private String name;
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "passport_id", referencedColumnName =
"id")
    private Passport passport;
}
@Entity
public class Passport {
    @Id
    @GeneratedValue
    private Long id;
    private String number;
}
```

1.1.5 Односторонній і двосторонній зв’язок @OneToOne

У JPA `OneToOne` може бути реалізований односторонньо або двосторонньо, залежно від того, скільки сутностей знає про зв’язок.

Односторонній `OneToOne`:

– тільки одна сутність містить посилання на іншу;

– інша сутність не має поля для зворотного зв’язку;

– у базі даних зазвичай створюється зовнішній ключ в таблиці власника зв’язку.

Два класи пов’язані відношеннями `@OneToOne` (ліст. 1.4):

– лише `User` знає про `Passport`;

– таблиця `User` має колонку `passport_id`;

– з `Passport` неможливо напряму отримати власника, якщо немає додаткового запиту.

Лістинг 1.4 – Класи пов’язані одностороннім зв’язком `@OneToOne`

```
@Entity
public class User {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
    @OneToOne
    @JoinColumn(name = "passport_id")
    private Passport passport;
}
@Entity
public class Passport {
    @Id
    @GeneratedValue
    private Long id;
    private String number;
}
```

Двосторонній `OneToOne`:

– обидві сутності містять посилання одна на одну;

– один з об’єктів є власником зв’язку (той, хто містить `@JoinColumn`), інший використовує `mappedBy`;

– дозволяє отримати доступ до іншого об’єкта з будь-якого боку зв’язку.

Два класи пов’язані відношеннями `@OneToOne` (ліст. 1.5):

– `User` – власник зв’язку (`@JoinColumn(name = "passport_id")`);

– `Passport` – зворотній кінець (`mappedBy = "passport"`);

– можна отримати користувача з паспорта: `passport.getUser()`.

Параметри анотацій зв’язку описані у табл. 1.1.

Лістинг 1.5 – Класи пов’язані двостороннім зв’язком `@OneToOne`

```
@Entity
public class User {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "passport_id")
    private Passport passport;
}
@Entity
public class Passport {
    @Id
    @GeneratedValue
    private Long id;
    private String number;
    @OneToOne(mappedBy = "passport")
    private User user;
}
```

Таблиця 1.1 – Параметри анотацій зв’язку

Параметр	Опис
`mappedBy`	вказує поле у зв’язаній сутності, що є власником зв’язку
`cascade`	каскадні операції (`PERSIST`, `MERGE`, `REMOVE`, `ALL`)
`fetch`	стратегія завантаження: `LAZY` (відкладене), `EAGER` (одразу)
`orphanRemoval`	видалення «сиріт», що більше не належать колекції
`@JoinColumn(name, referencedColumnName)`	ім’я колонки зовнішнього ключа і колонка в іншій таблиці
`@JoinTable(name, joinColumns, inverseJoinColumns)`	параметри проміжної таблиці для `ManyToMany`

1.1.6 Стратегії завантаження пов’язаних об’єктів (FetchType)

Коли ми працюємо зі зв’язаними сутностями у Spring Data JPA, дуже важливо розуміти, коли і як підвантажуються дані з бази. Саме для цього використовується механізм фетчингу (fetching), який визначає стратегію завантаження пов’язаних об’єктів.

У JPA існують два основні типи фетчингу: `LAZY` і `EAGER`.

Тип `FetchType.LAZY` означає, що пов’язані об’єкти або колекції не підвантажуються відразу разом із головною сутністю. Вони завантажуються лише у той момент, коли до них дійсно звертаються в коді. Це дозволяє економити ресурси і уникати зайвих SQL-запитів, особливо коли колекції великі або не завжди потрібні. Наприклад, якщо у нас є користувач із сотнями замовлень, немає сенсу витягати всі замовлення одразу, якщо ми хочемо просто вивести ім’я користувача:

```
@OneToMany(mappedBy = "user", fetch = FetchType.LAZY)
private List<Order> orders;
```

Тип `FetchType.EAGER`, навпаки, підвантажує всі пов’язані об’єкти одразу разом із головною сутністю. Це зручно, коли пов’язані дані завжди потрібні і їх варто отримати одним запитом. Наприклад, якщо користувач завжди має один паспорт і ми хочемо показати дані паспорта разом із користувачем:

```
@OneToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "passport_id")
private Passport passport;
```

Важливо пам’ятати, що надмірне використання `EAGER`-фетчингу може призвести до проблем з продуктивністю, особливо при великих колекціях або складних двосторонніх зв’язках. З іншого боку, `LAZY`-фетчинг може викликати

проблему N+1 запитів, якщо не контролювати підвантаження пов’язаних об’єктів за допомогою JOIN FETCH або кастомних запитів.

Таким чином, вибір стратегії фетчингу залежить від конкретної задачі: `LAZY` підходить для великих або необов’язкових колекцій, `EAGER` – для невеликих і завжди потрібних пов’язаних об’єктів. Розуміння цих нюансів допомагає писати ефективні додатки з мінімальним навантаженням на базу даних та без зайвих запитів.

1.1.7 Каскадні операції у JPA (CascadeType)

У реальних додатках часто трапляється ситуація, коли одна сутність тісно пов’язана з іншою: наприклад, користувач має список замовлень, або замовлення завжди має прив’язаний платіж. У таких випадках було б дуже незручно вручну виконувати операції з усіма пов’язаними об’єктами щоразу – зберігати, оновлювати або видаляти їх окремо. Саме для цього у JPA існує механізм каскадних операцій (CascadeType), який дозволяє автоматично «поширювати» певні дії з однієї сутності на пов’язані об’єкти.

Каскадні операції визначають, які дії над головною сутністю будуть автоматично застосовуватися до її зв’язаних об’єктів. Наприклад:

– `PERSIST` – якщо ми зберігаємо користувача, всі його замовлення теж збережуться автоматично;

– `MERGE` – при оновленні стану користувача, оновлюються і пов’язані замовлення;

– `REMOVE` – видалення користувача видаляє його замовлення з бази;

– `ALL` – всі перелічені операції застосовуються одночасно.

Уявіть просту ситуацію: ви створюєте нового користувача з кількома замовленнями. Без каскаду потрібно спочатку зберегти користувача, потім пройтися по кожному замовленню і викликати `save()` окремо (ліст. 1.6).

Лістинг 1.6 – Код створення нового користувача і додавання замовлення

```
User user = new User();
user.setName("Ivan");
Order order1 = new Order();
order1.setProduct("Laptop");
order1.setUser(user);
Order order2 = new Order();
order2.setProduct("Phone");
order2.setUser(user);
user.getOrders().add(order1);
user.getOrders().add(order2);
userRepository.save(user); // Всі замовлення збережуться
автоматично
```

Каскад дозволяє зробити все за один раз, зберігаючи логіку цілісності даних і скорочуючи код (ліст. 1.7). Завдяки `cascade = CascadeType.ALL` нам не потрібно окремо зберігати `order1` та `order2`.

Лістинг 1.7 – Приклад використання каскадних операцій

```
@Entity
public class User {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL,
orphanRemoval = true)
    private List<Order> orders = new ArrayList<>();
}
@Entity
public class Order {
    @Id
```

Кінець лістингу 1.7

```
    @GeneratedValue
    private Long id;
    private String product;
    @ManyToOne
    private User user;
}
```

Параметр `orphanRemoval = true` додатково забезпечує, що якщо замовлення видалити з колекції користувача, воно також буде видалене з бази:

```
user.getOrders().remove(order1);
userRepository.save(user); // order1 автоматично видаляється
```

1.1.8 Кастомні запити для роботи зі зв’язаними таблицями у JPA

У Spring Data JPA стандартні методи репозиторіїв (`findById`, `findAll`, `save`) зручні для простих CRUD-операцій, але часто у реальних застосунках виникає потреба отримати дані за складними умовами або підвантажити пов’язані об’єкти одразу. Для цього використовуються кастомні запити.

Кастомні запити – це запити, які ми самостійно визначаємо в репозиторії для отримання потрібної інформації. Вони можуть бути реалізовані трьома основними способами:

– JPQL (Java Persistence Query Language) – об’єктно-орієнтована мова запитів, схожа на SQL, але працює із сутностями, а не таблицями;

– нативні SQL-запити – звичайні SQL-запити до бази, зручні, коли потрібна складна оптимізація або специфічні конструкції;

– методи-генератори – автоматична генерація запитів на основі імен методів репозиторію (`findBy`, `findByOrdersProduct` тощо).

Кастомні запити потрібні для:

– підвантаження зв’язаних даних разом із головною сутністю (`JOIN FETCH`);

– вибірки за складними умовами, наприклад: всі замовлення користувача за певний період або товари з певною категорією;

– уникнення N+1 проблеми, коли при `LAZY`-завантаженні колекцій створюється занадто багато SQL-запитів.

Розглянемо приклад JPQL. Уявімо, що у нас є користувачі та їх замовлення (`User` та `Order`). Ми хочемо отримати користувача разом з усіма замовленнями одним запитом:

```
public interface UserRepository extends JpaRepository<User,
Long> {
@Query("SELECT u FROM User u JOIN FETCH u.orders WHERE u.id
= :id")
User findUserWithOrders(@Param("id") Long id);
}
```

`JOIN FETCH` дозволяє підвантажити всі замовлення (`orders`) разом із користувачем. Без `FETCH` при `LAZY`-завантаженні колекції замовлень підвантажуються окремим запитом на кожне звернення, що може створити N+1 проблему.

Розглянемо приклад приклад нативного SQL-запиту. Іноді потрібна оптимізація або специфічний SQL, наприклад сортування або обмеження кількості рядків. У цьому прикладі ми використовуємо реальні назви таблиць і колонок. Параметр `nativeQuery = true` говорить JPA, що це чистий SQL**,** а не JPQL:

```
@Query(value = "SELECT * FROM users u JOIN orders o ON u.id =
o.user_id WHERE o.product = :product", nativeQuery = true)
List<User> findUsersByOrderedProduct(@Param("product") String
product);
```

Стосовно методів-генераторів репозиторію, то JPA дозволяє автоматично створювати запити на основі імен методів. Перевага – швидка розробка без написання SQL/JPQL, недолік – складні умови може бути важко реалізувати у методах-генераторах. Наприклад:

```
List<User> findByOrdersProduct(String product);
```

JPA розпізнає, що треба виконати `JOIN` з таблицею `orders` і відфільтрувати за полем `product`.

1.1.9 Обробка видалення і оновлення зв’язаних об’єктів

Обробка видалення та оновлення визначає, як змінюються або видаляються елементи колекцій або зв’язаних об’єктів. Це потрібно для: контролю цілісності даних при зміні колекцій або видаленні сутностей та уникнення «сиріт» (об’єктів, що більше не належать жодній сутності).

Приклад видалення елемента з колекції:

```
User user = userRepository.findById(1L).get();
Order order = user.getOrders().get(0);
user.getOrders().remove(order);
userRepository.save(user);
```

Приклад оновлення колекції:

```
Order newOrder = new Order();
newOrder.setProduct("Laptop");
newOrder.setUser(user);
user.getOrders().add(newOrder);
userRepository.save(user);
```

1.1.10 Бібліотека Lombok

Lombok – це спеціальна бібліотека для Java, яка створена для того, щоб позбавити розробника від рутинного та повторюваного коду. У типовому проєкті на Java досить часто доводиться писати гетери, сетери, конструктори, методи equals, hashCode чи toString, навіть коли вони не несуть жодної бізнес-логіки. Такі методи займають багато місця й роблять клас важчим для читання. Lombok вирішує цю проблему, додаючи у код спеціальні анотації, які в процесі компіляції «підкладають» потрібні методи автоматично. У результаті виходить, що вихідний код виглядає максимально компактно, але при цьому в байткоді клас має усі необхідні методи.

Уявімо собі просту модель користувача, яка містить лише кілька полів: ім’я, електронну адресу та вік. Без Lombok довелося б написати гетери й сетери для кожного поля, конструктори для ініціалізації об’єкта та, можливо, метод toString для зручного відображення. Це щонайменше кілька десятків рядків коду, які дублюють один і той самий шаблон. А тепер подивимося, як виглядає той самий клас із Lombok:

```
import lombok.Data;

@Data
public class User {
private String name;
private String email;
private int age;
}
```

Анотація @Data автоматично генерує для нас усі гетери та сетери, equals, hashCode, toString, а також конструктор для всіх полів. Вихідний код залишається надзвичайно лаконічним – лише три поля та одна анотація. Але під

час виконання ми отримаємо клас із повним набором потрібних методів, які працюють так, ніби ми написали їх вручну.

Таким чином, `Lombok` можна назвати інструментом, який «очищає» код від шаблонної рутини, залишаючи розробникові більше простору для справжньої логіки. Він підвищує читабельність, зменшує кількість помилок, що виникають через копіювання-кодування, і робить проєкти більш підтримуваними. Це бібліотека, яка не змінює саму Java, але робить її значно зручнішою для щоденного використання.

1.2 Хід роботи

Створіть базу даних MySQL, яка буде мати дві сутності: країну та місто. В одній країні може бути кілька міст (відношення «один-до-багатьох»).

Схема даних повинна виглядати наступним чином:

```
CREATE TABLE IF NOT EXISTS `country` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;

CREATE TABLE IF NOT EXISTS `city` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `country_id` int(11) NOT NULL,
  `name` varchar(255) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `country_id` (`country_id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;
```

Ми хочемо виконати типові дії над цими сутностями: перегляд всього списку, пошук по `id`, додавання, оновлення і видалення записів.

Створіть типовий проєкт Spring Boot, наприклад, використовуючи https://start.spring.io/. Додайте необхідні залежності в файл pom.xml в корені проєкту в розділі dependencies.

Стандартна залежність від `spring-boot-starter-web` для роботи з будь-якими REST додатками (потрібна для функціональності `rest` контролера):

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

Залежності для роботи з Spring Data JPA (модуль Spring для роботи з базою даних):

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

Залежності для роботи з MySQL:

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

Залежність, щоб мати можливість явно використовувати анотації `@JsonIgnore` бібліотеки `Jackson`**:**

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
    <version>2.20</version>
</dependency>
```

Залежність для пакету `lombok`, який спрощує та автоматизує створення геттерів та сеттерів у класах `POJO`, видаляючи велику кількість шаблонного коду:

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.40</version>
</dependency>
```

Налаштування підключення до бази даних зробіть як в попередніх лабораторних роботах. Вміст файлу /resources/application.properties буде виглядати як на ліст. 1.8. Стандартний код для класу `main` див. ліст. 1.9.

Лістинг 1.8 – Вміст файлу конфігурації application.properties

```
spring.application.name=rest_api_nested_tables
# ---DataSource (MySQL) ---
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
# JPA / debug
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Лістинг 1.9 – Код класу main (стандартний)

```
package com.csn.rest_api_nested_tables;
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class RestApiSpringIoApplication {
     public static void main(String[] args) {
     SpringApplication.run(RestApiSpringIoApplication.class,
args);
     }
}
```

Створіть модуль моделі, в якому будуть зберігатися моделі проєкту.

У Hibernate клас, який співставляє поля `POJO` класу на таблицю в базі даних, називається класом сутності (entity). Для таблиці `country` такий клас буде виглядати так як на ліст. 1.10.

Лістинг 1.10 – Код `POJO` класу `Country`

```
package com.csn.rest_api_nested_tables.model;

import jakarta.persistence.*;
import lombok.Data;
import java.util.List;

@Entity
@Data
@Table(name = "country")
public class Country {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
```

Кінець лістингу 1.10

```
    private String name;
    @OneToMany(mappedBy = "country", cascade =
CascadeType.ALL, orphanRemoval = true)
    private List<City> cities;
}
```

Анотація `@Entity` сигналізує Hibernate, що даний клас є сутністю, яку потрібно зберігати в базі даних.

Анотація `@Table` дозволяє задати конкретне ім’я таблиці, з якою пов’язана ця сутність. Якщо назви класу та таблиці збігаються, використання `@Table` не є обов’язковим.

Анотація `@Id` позначає поле як унікальний ідентифікатор сутності. За допомогою `@GeneratedValue` можна вибрати спосіб автоматичної генерації цього ідентифікатора.

У нашому випадку застосовується `GenerationType.IDENTITY`, оскільки на рівні бази даних використовується послідовність, яка автоматично заповнює значення під час вставки нового запису.

Анотація `@OneToMany` визначає зв’язок «один-до-багатьох» між сутностями, наприклад, між країною та містом, коли в одній країні може бути кілька міст. Тому вона застосовується до колекційних полів, наприклад, списків.

Параметр `mappedBy` вказує на поле в сутності `City`, яке посилається на свій батьківський об’єкт (`Country`).

Параметр `orphanRemoval` означає, що при видаленні батьківського об’єкта всі «осиротілі» дочірні об’єкти також видаляються.

Анотація `@Data` з бібліотеки `Lombok` автоматично генерує геттери, сеттери та інші допоміжні методи для всіх полів класу.

У сутності `City` використовується зворотній зв’язок (ліст. 1.11)

.

Лістинг 1.11 – Код POJO класу City

```
package com.csn.rest_api_nested_tables.model;

import com.fasterxml.jackson.annotation.JsonIgnore;
import jakarta.persistence.*;
import lombok.Data;

@Entity
@Data
@Table(name = "city")
public class City {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String name;
    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "country_id")
    private Country country;
    @JsonIgnore
    public Country getCountry() {
        return country;
    }
}
```

Анотація `@ManyToOne` є оберненим зв'язком до `@OneToMany` і застосовується в дочірній сутності. Анотація `@JoinColumn` визначає ім'я зовнішнього ключа в базі даних, який використовується для зв'язку з батьківською сутністю.

Зверніть увагу, що метод `get` для поля `country` має анотацію `@JsonIgnore`. Без неї при серіалізації вкладених міст у формат JSON утворювався би дуже великий JSON, оскільки об'єкти посилалися б один на

одного. Щоб бачити міста вкладеними у країну, ми просто приховуємо інформацію про країну в дочірніх сутностях.

Далі створюємо модуль репозиторію та реалізуємо в ньому сервіс для роботи з базою даних (ліст. 1.12).

Лістинг 1.12 – Код класу `CountryDao`

```
@Repository
@Transactional
public class CountryDao {
    @PersistenceContext
    private EntityManager entityManager;
    public List<Country> getAll() {
        return entityManager.createQuery("from Country c
order by c.id desc", Country.class).getResultList();
    }
    public Country getById(int id) {
        return entityManager.find(Country.class, id);
    }
}
```

Анотація `@Repository` позначає клас як компонент для взаємодії з базою даних.

Анотація `@Transactional` робить усі публічні методи транзакційними, забезпечуючи атомарність операцій. `@PersistenceContext` надає доступ до EntityManager у контексті виконання – сервісу, який дозволяє виконувати основні операції з сутностями.

Перший спосіб роботи з даними передбачає створення запитів, схожих на SQL, проте це HQL (Hibernate Query Language). На відміну від SQL, HQL оперує об’єктами та сутностями у стилі ООП. Для пошуку за `id` застосовується стандартний метод `find()`, який приймає тип сутності та значення ідентифікатора.

Далі створюємо модуль контролера та реалізуємо у ньому REST-контролер, який обробляє запити у форматі JSON (ліст. 1.13).

Лістинг 1.13 – Код класу контролера `GeoController`

```
@RestController
@RequestMapping(value = "/api", produces =
MediaType.APPLICATION_JSON_VALUE)
public class GeoController {
    @Autowired
    private CountryDao countryDao;
    @GetMapping("/countries")
    public List<Country> getAllCountries() {
        return countryDao.getAll();
    }
    @GetMapping("/countries/{countryId}")
    public Country getCountryById(@PathVariable("countryId")
int id) {
        return countryDao.getById(id);
    }
}
```

Анотація `@RequestMapping` задає базовий URL для REST-запитів. Параметр `produces` вказує, що відповіді від контролера будуть у форматі JSON. За допомогою `@Autowired` підключається наш репозиторій для роботи з базою даних. Варто зазначити, що у реальних проєктах між контролером і репозиторієм зазвичай присутній сервісний шар із анотацією `@Service`, де реалізується вся бізнес-логіка. У нашому випадку цей шар опущено для спрощення, оскільки контролер лише перенаправляє запити до репозиторію.

Анотація `@GetMapping` позначає методи для обробки GET-запитів, а `@PathVariable` дозволяє використовувати частину URL як параметр запиту (наприклад, числовий ID країни).

Для створення нового запису в базі даних додамо метод у наш репозиторій. Для цього додайте наступний код у кінець класу `CountryDao`:

```
public Country create(Country country) {
    for (City city : country.getCities()) {
        city.setCountry(country);
    }
    entityManager.persist(country);
    return country;
}
```

Оскільки разом із країною надходять вкладені сутності міст (лише з їхніми назвами), спочатку необхідно прив’язати кожне нове місто до цієї країни. Після цього ми викликаємо метод `persist()` для батьківської сутності. Він використовується лише для створення нових записів, і завдяки правильно налаштованим анотаціям Hibernate автоматично збереже всі дочірні об’єкти разом із країною.

У контролері (`GeoController`) для цього передбачимо окремий метод:

```
@PostMapping("/countries")
@ResponseStatus(HttpStatus.CREATED)
public Country createCountry(@RequestBody Country country) {
    return countryDao.create(country);
}
```

Анотація `@PostMapping` визначає метод, який буде обробляти HTTP-запити типу POST, а `@ResponseStatus` задає код відповіді, який повернеться клієнту. Для створення нових ресурсів у REST-архітектурі коректно використовувати код 201 Created, а не стандартний 200 OK.

За допомогою `@RequestBody` ми вказуємо, що тіло запиту необхідно автоматично перетворити у відповідний клас Java – у нашому випадку це `Country`.

Тепер ми можемо додати нову країну разом із вкладеними містами. Для цього запускаємо застосунок і надсилаємо POST-запит на адресу:

```
{
   "name": "Ukraine",
   "cities":[{"name":"Kyiv"}]
}
```

Метод підтримує передачу списку міст, тому для додавання декількох міст одразу можна записати:

```
{
   "name": "Germany",
   "cities":[{"name":"Berlin"}, {"name":"Cologne"},
{"name":"Bern"}]
}
```

Зверніть увагу, що під час додавання ми передаємо лише імена для обох сутностей. У відповіді ж отримаємо ту ж структуру, але вже з автоматично згенерованими `id`, які Hibernate призначає новим записам.

Якщо операція виконана успішно, у таблицях бази даних з’являться нові записи, а метод `POST` поверне HTTP 201 – Created.

Для того щоб отримати список усіх країн разом із містами, достатньо надіслати `GET`-запит на адресу: http://127.0.0.1:8080/api/countries/.

Розберемося, як відбувається оновлення записів. Припустимо, що при оновленні ми можемо змінити назву країни, і кожен раз будемо видаляти і додавати нові міста. Метод в репозиторії буде виглядати як у ліст. 1.14.

Лістинг 1.14 – Приклад методу для оновлення даних

```
public Country update(int id, Country country) {
    Country original = entityManager.find(Country.class, id);
    if (original != null) {
        original.setName(country.getName());
        for (City city : original.getCities()) {
            entityManager.remove(city);
        }
        original.getCities().clear();
        for (City city : country.getCities()) {
            city.setCountry(original);
            original.getCities().add(city);
            entityManager.persist(city);
        }
        entityManager.merge(original);
    }
    return original;
}
```

Спершу ми намагаємося знайти у базі даних країну разом з її містами за переданим ідентифікатором. Якщо такий запис існує, то змінюємо назву країни на нову. Далі послідовно перебираємо всі наявні міста та видаляємо їх через `entityManager.remove()`. Тут важливо розуміти: достатньо передати сам об’єкт, і Hibernate сам подбає про видалення відповідного рядка в таблиці.

Після цього очищуємо колекцію міст у країні та додаємо ті, що прийшли у тілі запиту. Для кожного з них виконується виклик `persist()`, щоб зберегти їх у базі. Наприкінці застосовуємо метод `merge()` для батьківської сутності – він підходить саме для оновлення вже наявних даних.

У контролері (клас `GeoController`) тепер можна створити метод, який оброблятиме такі запити на оновлення:

```
@PutMapping("/countries/{countryId}")
public Country updateCountry(@PathVariable("countryId") int
id, @RequestBody Country country) {
    return countryDao.update(id, country);
}
```

У більшості REST-архітектур для зміни вже існуючих даних застосовується метод `PUT`, що й позначається анотацією `@PutMapping`. Формат JSON у тілі запиту для оновлення повністю повторює структуру, яка використовується під час створення нового запису. Водночас сам запит потрібно надсилати на адресу, де в URL додатково вказується ідентифікатор країни, яку потрібно змінити. Наприклад: http://127.0.0.1:8080/api/countries/27.

Розберемося, як відбувається видалення даних. Щоб завершити цикл життя сутності, нам залишається реалізувати операцію видалення. У шарі репозиторію вона виглядатиме як у ліст. 1.15. Тут ми спочатку намагаємося знайти існуючий запис за її `id`, а потім видаляємо її за допомогою вже знайомого нам способу. При цьому будуть видалені і міста, пов’язані з країною.

Лістинг 1.15 – Приклад реалізації операції видалення

```
public void delete(int id) {
    Country country = entityManager.find(Country.class, id);
    if (country != null) {
        entityManager.remove(country);
    }
}
```

Метод на рівні контролера (ліст. 1.16). Анотація `@DeleteMapping` використовується для позначення методу, що реагує на HTTP-запити типу `DELETE`. У поєднанні з `@ResponseStatus(HttpStatus.NO_CONTENT)` це означає, що після успішного виконання запиту клієнт отримає відповідь із кодом 204 – No Content, а тіло відповіді залишиться порожнім.

Лістинг 1.16 – Приклад реалізації операції видалення на рівні контролеру

```
@DeleteMapping("/countries/{countryId}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteCountry(@PathVariable("countryId") int id)
{
    countryDao.delete(id);
}
```

Фінальну структуру проєкта можемо бачити на рис. 1.1. Завдяки використанню Hibernate нам вдалося реалізувати базові операції з сутностями практично в кілька рядків коду. А інтеграція зі Spring Boot дала можливість оформити все відповідно до архітектури REST. Анотації `@OneToMany` та `@ManyToOne` описують взаємозв'язки між класами, що дозволяє зручно керувати цілими групами пов'язаних об'єктів у межах одного методу.

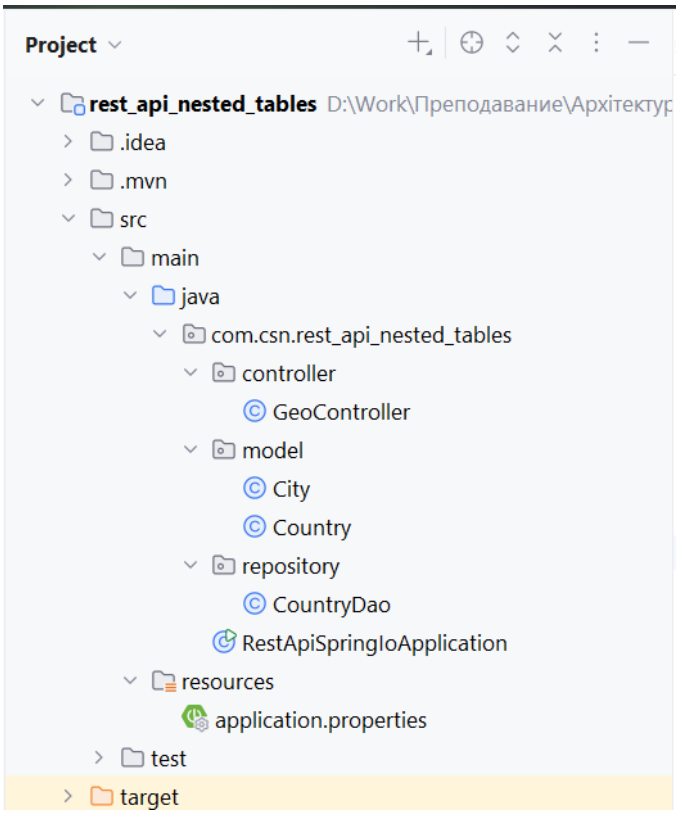

Рисунок 1.1 - Фінальна структура проєкта

Крім того, Hibernate бере на себе роботу з формуванням SQL-запитів, приховуючи технічні деталі. Завдяки цьому код стає зрозумілішим і не прив'язується до конкретної СУБД. Наприклад, ви можете без особливих зусиль перейти з PostgreSQL на MySQL чи Oracle, змінивши лише кілька параметрів у конфігураційних файлах.

1.3 Завдання до лабораторної роботи

Впровадити методи CRUD для роботи з містами певної країни, щоб вони використовували префікс URL на кшталт цього: /api/countries/{country_id}/.

Поточна реалізація не перевіряє унікальність назв країн і міст у межах країни. Додайте унікальний індекс в базу даних в поле назви для таблиці країн і контроль в коді, що якщо країна з такою назвою вже існує, потрібно додавати міста в існуючу країну, а не створювати новий запис в таблиці країн. Так само назви міст у межах однієї країни мають бути унікальними. Якщо ви додаєте місто, таке місто вже існує в країні, ви повинні запобігти створенню дублікатів.

В окремому проєкті створити повноцінний REST API для роботи з базою даних музичного сервісу. Всі дані повинні зберігатися в базі даних MySQL. Сутності доменної моделі:

– артист (`artist`) з полями: `id`, `name`, `title` (text), `country`;

– альбоми (`albums`) з наступними полями: `id`, `artist_id`, `title`, `year`, `info` (text);

– пісні (`songs`) з полями: `id`, `album_id`, `title`, `duration` (int), `genre`.

Необхідно впровадити операції CRUD для всіх об'єктів і додаткові операції:

– `findArtistByCountry(country)` – для пошуку виконавців по країні;

– `findArtistSongs(song_title)` – для відображення списку всіх пісень виконавця (інформація про альбом для кожної пісні повинна бути присутня в списку полів виведення);

– `findSongsByGenre(genre)` – для пошуку пісень за жанром (у полях виводу повинні бути присутніми дані про альбом та виконавця).

1.4 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.

1.5 Контрольні питання

1. Що таке зв’язки між таблицями у базі даних і які основні типи зв’язків підтримує JPA (`One-to-One`, `One-to-Many`, `Many-to-Many`)?

2. Як у Spring Boot за допомогою JPA реалізувати зв’язок «один-до-багатьох» між сутностями?

3. Яку роль відіграє анотація `@JoinColumn` у визначенні зв’язків між таблицями?

4. Чим відрізняється використання `@OneToMany(mappedBy = ...)` від використання `@JoinColumn`?

5. Як правильно серіалізувати сутності з двосторонніми зв’язками, щоб уникнути рекурсії у JSON-відповідях (наприклад, за допомогою `@JsonManagedReference` та `@JsonBackReference`)?

6. Яким чином у REST API реалізується створення сутності разом із пов’язаними об’єктами (наприклад, замовлення з товарами)?

7. Які підходи існують для завантаження пов’язаних даних: ліниве (`Lazy`) та жадібне (`Eager`) завантаження, і в чому їх відмінність?

8. Як побудувати REST-контролер для отримання об’єкта разом із його пов’язаними сутностями, наприклад, користувача разом зі списком його замовлень?

ЛАБОРАТОРНА РОБОТА № 6 – ТЕСТУВАННЯ REST API

Мета роботи: одержати знання та навички з написання Unit та інтеграційних тестів для REST API за допомогою бібліотек JUnit, Mockito та MockMVC [1-3, 7-9].

2.1 Теоретичні відомості

Тестування монолітного застосунку та мікросервісної архітектури відрізняється перш за все підходом до організації та охоплення тестів. У монолітному застосунку, де всі компоненти живуть у єдиному процесі, легко покрити взаємодію між модулями за допомогою інтеграційних тестів, а запуск End-to-End сценаріїв зазвичай стабільний і передбачуваний. Тут тестувальник або розробник має повний контроль над середовищем: можна легко підготувати тестові дані, відтворити стан бази і перевірити взаємодію між компонентами без додаткових хитрощів.

У мікросервісах ситуація значно складніша. Кожен сервіс існує автономно, взаємодіє з іншими через API, тому навіть простий сценарій інтеграції потребує емуляції або імітації зовнішніх залежностей. Часто доводиться використовувати mock-и, stub-и або тестові контейнери, щоб забезпечити стабільність тестів і уникнути непередбачуваних збоїв через мережеві виклики. При цьому важливо перевіряти не лише функціональність окремого сервісу, а й контрактні взаємодії між сервісами – чи відповідає API очікуваним параметрам і даним, та як система поводиться при помилках або недоступності певного сервісу.

Також у мікросервісах значно складніше організувати End-to-End тести. Тут надійність тестів залежить від стану кількох сервісів одночасно, від мережевих затримок і від того, наскільки добре змодельоване середовище. Через це тести стають довшими і вимагають ретельної підготовки, але водночас вони більш реалістично відображають поведінку системи в продакшн-середовищі.

У результаті підхід до тестування стає різним: моноліти дозволяють швидко й відносно просто перевіряти всю систему в цілому, тоді як у мікросервісах тестування більше нагадує побудову ланцюжка перевірок для кожного сервісу та його взаємодій, що вимагає стратегічного планування та більшої уваги до деталей.

2.2.1 Види тестування мікросервісів

Розглянемо деякі види тестування мікросервісів:

– unit-тестування;

– integration-тестування;

– contract-тестування;

– end-to-end (E2E) тестування;

– тестування на відмовостійкість (resilience / chaos testing);

– performance і load-тестування.

Unit-тести перевіряють логіку окремого сервісу на рівні класів і методів, ізольовано від зовнішніх залежностей. Мета: впевнитися, що внутрішні методи працюють правильно. Як реалізується: зазвичай через mock-и для зовнішніх викликів, бази даних або інших сервісів. Приклад: тест методу сервісу, який обчислює суму замовлень користувача, незалежно від того, що відбувається в базі чи в інших сервісах.

Integration-тести перевіряють взаємодію компонентів всередині одного сервісу або між сервісом і його зовнішніми залежностями, наприклад базою даних або чергами повідомлень. Мета: переконатися, що сервіс коректно працює з реальними компонентами. Особливість: у мікросервісах часто використовують тестові контейнери або in-memory бази для ізоляції тестового середовища. Приклад: сервіс створює замовлення та записує його у базу даних; тест перевіряє, що всі поля коректно збережені.

Contract-тести перевіряють, що інтерфейси між сервісами відповідають домовленостям (контрактам). Мета: запобігти помилкам у взаємодії різних

сервісів через API. Як реалізується: перевіряється структура запитів/відповідей, формати полів, очікувані статуси HTTP. Приклад: сервіс оплати очікує, що сервіс замовлень віддасть список замовлень у певному JSON-форматі; contract-тест гарантує, що зміни в сервісі замовлень не порушать роботу оплати.

E2E-тести перевіряють повну роботу системи як єдиного цілого, включно з усіма мікросервісами, мережею, базами даних та зовнішніми інтеграціями. Мета: впевнитися, що бізнес-процеси працюють правильно для користувача. Складність: потребує підготовки середовища, де запущені всі сервіси, або використання емуляцій/моків для деяких компонентів. Приклад: користувач створює замовлення, сервіс замовлень зберігає його, сервіс оплати обробляє платіж, сервіс доставки отримує інформацію про замовлення, E2E-тест перевіряє весь сценарій від початку до кінця.

Тестування на відмовостійкість (resilience / chaos testing) перевіряє, як система поводиться при збоях одного або кількох сервісів. Мета: оцінити стабільність, правильність обробки помилок та відновлення після відмов. Методи: введення затримок, штучних помилок, відключення окремих сервісів. Приклад: якщо сервіс оплати тимчасово недоступний, замовлення повинно бути збережене в черзі, щоб обробка продовжилася після відновлення.

Performance і load-тестування (тестування продуктивності) перевіряє, як сервіси працюють під навантаженням. Мета: оцінити швидкість відповіді, витрати ресурсів і масштабованість. Особливість: у мікросервісній архітектурі важливо тестувати не лише окремий сервіс, а й їхню взаємодію при високій кількості одночасних викликів. Приклад: перевірка, чи витримує сервіс замовлень 1000 запитів на створення замовлень за хвилину без падіння продуктивності.

В рамках цієї лабораторної роботи ми зосередимося на unit-тестах та integration-тестах для мікросервісів.

Unit-тести у мікросервісній архітектурі виконують дуже важливу роль, але часто їх неправильно застосовують. Основна мета unit-тестів – швидко і надійно перевіряти внутрішню логіку окремого сервісу без залежності від зовнішніх

систем. Вони не повинні перевіряти взаємодію з іншими сервісами, мережею чи базою даних – для цього існують інтеграційні та контрактні тести.

Unit-тести потрібні для:

– перевірки бізнес-логіки сервісу: вони дозволяють переконатися, що методи сервісу працюють правильно для різних вхідних даних і обробляють усі граничні випадки (наприклад, метод обчислення вартості замовлення повинен правильно застосовувати знижки, незалежно від того, чи є підключена база чи інші сервіси);

– виявлення помилок на ранньому етапі: тести ізольовані від зовнішніх залежностей, їх можна запускати дуже швидко під час розробки, що дозволяє виявляти помилки одразу, ще до інтеграції з іншими сервісами;

– документування поведінки коду: вони одночасно показують, як сервіс очікує отримувати дані і що робити у різних ситуаціях, допомагаючи новим розробникам швидко зрозуміти бізнес-логіку.

Unit-тести не треба застосовувати для тестування: взаємодії з іншими мікросервісами; реальних викликів до бази даних; складної конфігурації середовища.

Unit-тести повинні бути швидкими, ізольованими і надійними. Кожен тест перевіряє одну функцію або метод у сервісі, використовуючи підставні об’єкти (mocks, stubs) для всіх зовнішніх залежностей. Це дозволяє запускати сотні тестів за секунди під час розробки, не чекаючи підняття всього середовища мікросервісів.

Інтеграційні тести у мікросервісній архітектурі покликані перевіряти взаємодію сервісу з його реальними залежностями, такими як база даних, інші сервіси через API, черги повідомлень чи кеш. Якщо unit-тести перевіряють внутрішню логіку окремого класу, то інтеграційні тести демонструють, що всі компоненти сервісу працюють разом коректно.

Інтеграційні тести потрібні для:

– перевірки взаємодії з базою даних: вони показують, що сервіс правильно читає і записує дані, застосовує транзакції та обробляє граничні сценарії

(наприклад, створення замовлення в базі через сервіс, перевірка, що всі поля збережені правильно, і зв’язані сутності підвантажені);

– тестування взаємодії з іншими сервісами або API: перевіряється, чи сервіс коректно формує запити до зовнішніх сервісів і обробляє їх відповіді. Для стабільності часто використовують mocks, stubs або тестові контейнери, які імітують поведінку реальних залежностей;

– виявлення проблем інтеграції (помилки конфігурації, неправильні трансформації даних або невідповідність контрактів між сервісами);

– тестування транзакцій та каскадних операцій.

Інтеграційні тести не треба застосовувати для тестування: логіки окремих методів; поведінки інших сервісів, якщо вони не ключові для сценарію; навантаження або стресу.

Інтеграційні тести мають бути повільнішими за unit-тести, бо залучають реальні компоненти, але вони не повинні перевіряти все підряд. Краще писати окремі тести для ключових сценаріїв: збереження і читання даних із бази; взаємодія з іншими сервісами через API; перевірка каскадних операцій і транзакцій.

Завдяки цьому інтеграційні тести дозволяють впевнено стверджувати, що сервіс коректно працює у середовищі, наближеному до реального, не покладаючись на ізоляцію, як у unit-тестах.

2.2.2 Бібліотека JUnit

JUnit – це бібліотека для автоматизованого тестування Java-коду. Вона використовується для перевірки правильності роботи окремих методів і класів, швидкого виявлення помилок та підтримки стабільності додатку під час розробки. Завдяки JUnit розробник може запускати тести багаторазово, отримувати зручні звіти і інтегрувати тести у CI/CD процеси. Основні анотації JUnit та їх призначення див. табл. 2.1, приклад використання – див. ліст. 2.1.

Таблиця 2.1 – Основні анотації JUnit та їх призначення

Анотація	Призначення
`@Test`	Позначає метод як тестовий, який буде виконаний JUnit
`@BeforeEach`	Виконується перед кожним тестом, наприклад для ініціалізації об’єктів
`@AfterEach`	Виконується після кожного тесту, наприклад для очищення ресурсів
`@BeforeAll`	Виконується один раз перед усіма тестами класу
`@AfterAll`	Виконується один раз після завершення всіх тестів класу
`@Disabled`	Тимчасово відключає тест, щоб він не виконувався
`@ParameterizedTest`	Дозволяє запускати один і той же тест з різними вхідними даними
`@ValueSource`	Використовується разом з `@ParameterizedTest` для передачі набору значень

Лістинг 2.1 – Приклад використання анотацій у unit-тестах

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
import static org.junit.jupiter.api.Assertions.assertEquals;
class CalculatorTest {
    private Calculator calculator;
    @BeforeEach
    void setup() {
        calculator = new Calculator();
    }
    @Test
    void additionTest() {
        int result = calculator.add(2, 3);
        assertEquals(5, result);
    }
}
```

Приклад використання анотацій у unit-тестах (ліст. 2.1) демонструє основну роботу JUnit: організацію тестів, підготовку середовища та перевірку очікуваного результату. Деякі пояснення коду:

– `@BeforeEach` ініціалізує об’єкт калькулятора перед кожним тестом;

– `@Test` позначає метод `additionTest` як тест;

– `assertEquals` перевіряє, що метод `add` повертає правильну суму.

2.2.3 Бібліотека Mockito

Mockito – це популярна бібліотека для створення підставних об’єктів (mock) у тестуванні Java-коду, особливо у unit-тестах. Вона дозволяє ізолювати перевірюваний код від зовнішніх залежностей, таких як бази даних, вебсервіси або інші класи, і перевіряти тільки логіку конкретного методу чи класу.

Розглянемо основні можливості бібліотеки Mockito.

Створення mock об’єктів. Mock об’єкт – це об’єкт-заглушка, який реалізує інтерфейс або клас, але не виконує реальної логіки. Він дозволяє контролювати поведінку залежностей під час тестування:

```
List<String> mockedList = Mockito.mock(List.class);
Mockito.when(mockedList.size()).thenReturn(5); // задаємо поведінку
```

Mockito дозволяє перевіряти (verification), які методи викликав код під тестуванням і скільки разів:

```
mockedList.add("test");
Mockito.verify(mockedList).add("test"); // перевірка виклику
```

Імітація винятків і поведінки методів для перевірки обробки помилок:

```
Mockito.when(mockedList.get(0)).thenThrow(new
RuntimeException("error"));
```

Створення spy-об’єктів. Spy дозволяє об’єкту виконувати реальні методи, але одночасно контролювати окремі виклики:

```
List<String> list = new ArrayList<>();
List<String> spyList = Mockito.spy(list);
Mockito.doReturn(100).when(spyList).size();
```

Mockito легко інтегрується з JUnit 5 та JUnit 4, дозволяючи автоматично створювати mock об’єкти та спрощувати тести.

Приклад використання бібліотеки Mockito (ліст. 2.2) демонструє створення mock списку замість реального, визначення поведінки (`size()` повертає 3), перевірку використання для тестування саме mock методу.

Лістинг 2.2 – Приклад використання бібліотеки Mockito

```
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import java.util.List;
import static org.junit.jupiter.api.Assertions.assertEquals;
class UserServiceTest {
    @Test
    void testGetUserCount() {
        List<String> mockedList = Mockito.mock(List.class);
        Mockito.when(mockedList.size()).thenReturn(3);
        int count = mockedList.size();
        assertEquals(3, count);
        // перевіряємо виклик
        Mockito.verify(mockedList).size();
    }
}
```

2.2.4 Бібліотека для integration-тестів MockMvc

MockMvc – це інструмент Spring Framework для тестування контролерів веб-додатків (REST API) без підняття реального HTTP-сервера. Він дозволяє перевіряти маршрути, обробку запитів, статуси відповідей, формат JSON та інші аспекти взаємодії клієнта з контролером, при цьому всі тести виконуються швидко і в ізольованому середовищі.

Розглянемо основні можливості бібліотеки MockMvc.

Виконання HTTP-запитів без реального сервера: дозволяє відправляти `GET`, `POST`, `PUT`, `DELETE` та інші HTTP-запити до контролерів прямо в тестах та повертає об'єкт `ResultActions`, з яким можна працювати далі, тобто перевіряти статус, заголовки, тіло відповіді:

```
mockMvc.perform(get("/api/users/1"))
        .andExpect(status().isOk());
```

Перевірка статусів відповідей: чи контролер повернув очікуваний HTTP-код (200, 201, 404, 400 тощо):

```
.andExpect(status().isNotFound());
```

Перевірка тіла відповіді: перевіряти JSON-відповіді за допомогою `jsonPath`:

```
.andExpect(jsonPath("$.name").value("John"))
        .andExpect(jsonPath("$.age").value(30))
```

Відправка даних у запиті: можна передавати JSON у тілі `POST` або `PUT` запитів, використовуючи `content()` і задаючи `contentType`:

```
mockMvc.perform(post("/api/users")
        .contentType(MediaType.APPLICATION_JSON)
        .content("{\"name\":\"John\",\"age\":30}"))
        .andExpect(status().isCreated());
```

Перевірка заголовків та параметрів: чи контролер обробив потрібні HTTP-заголовки, куки або параметри запиту:

```
.andExpect(header().string("Location", "/api/users/1"));
```

Приклад інтеграційного тестування за допомогою MockMvc див. ліст. 2.3.

Лістинг 2.3 – Приклад інтеграційного тестування за допомогою MockMvc

```
import org.junit.jupiter.api.Test;
import
org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.WebMv
cTest;
import org.springframework.test.web.servlet.MockMvc;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBu
ilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatc
hers.*;

@WebMvcTest(UserController.class)
class UserControllerTest {
    @Autowired
    private MockMvc mockMvc;
```

Кінець лістингу 2.3

```
    @Test
    void getUserById_returnsUser() throws Exception {
        mockMvc.perform(get("/api/users/1"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.name").value("John"))
                .andExpect(jsonPath("$.age").value(30));
    }
}
```

2.2 Хід роботи

2.2.1 Підготовка до створення Unit-тестів

Unit-тести складають базовий рівень перевірки програмного забезпечення під час розробки. Вони дають змогу оцінювати окремі ділянки коду без впливу зовнішніх систем чи залежностей. Наприклад, можна протестувати метод сервісу, який розраховує знижку, не використовуючи при цьому базу даних, API інших сервісів або сторонні бібліотеки.

Головні цілі unit-тестування: гарантувати, що код тестується в ізоляції; перевірити коректність реалізації бізнес-логіки; забезпечити правильну обробку граничних випадків і виключних ситуацій.

2.2.2 Створення проєкту

Для початку створіть новий проєкт на Spring Boot, наприклад, скориставшись генератором за адресою https://start.spring.io/.

Додайте наступні залежності до файлу pom.xml (ліст. 2.4). Бібліотека Spring Boot Test: ймовірно, ця залежність вже включена у pom.xml, проте потрібно додати виняток для Mockito, щоб вона підключалася як окрема залежність, а не через Spring Boot Test.

Лістинг 2.4 – Залежності у файлі pom.xml

```
<!-- JUnit 5 -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <scope>test</scope>
</dependency>

<!-- Mockito -->
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <scope>test</scope>
</dependency>

<!-- Spring Boot Test -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
         <exclusions>
             <exclusion>
                 <groupId>org.mockito</groupId>
                 <artifactId>mockito-core</artifactId>
             </exclusion>
         </exclusions>
</dependency>
```

2.2.3 Приклад програми: дисконтний магазин

Розглянемо приклад мікросервісу для обробки знижок. Уявімо, що існує сервіс DiscountService, який обчислює розмір знижки залежно від категорії користувача та кількості придбаних товарів.

Створіть у проєкті модуль services за шляхом /src/main/java/<ваше_ім'я_пакета>/services/ та всередині нього додайте клас `DiscountService` (ліст. 2.5).

Лістинг 2.5 – Код класу сервісу DiscountService

```
package com.csn.rest_api_spring_io.servies;
import org.springframework.stereotype.Service;

@Service
public class DiscountService {
    public double calculateDiscount(String userCategory, int
itemCount) {
        if (itemCount <= 0) {
            throw new IllegalArgumentException("Item count
must be greater than 0");
        }
        switch (userCategory.toLowerCase()) {
            case "vip":
                return itemCount * 0.2; // 20% знижка
            case "regular":
                return itemCount > 5 ? itemCount * 0.1 : 0.0;
// 10% знижка, якщо покупок > 5
            default:
                return 0.0; // Без знижки
        }
    }
}
```

Крок 1. Налаштуйте тестовий клас.

У IDE IntelliJ IDEA натисніть Ctrl+Shift+T, перебуваючи у вкладці з класом `DiscountService`. Повинно з'явитися спливаюче вікно, що дозволяє створити тест для даного класу (рис. 2.1). У вікні, що з'явилося, вибираємо Junit 5 (рис. 2.2). Клас `DiscountServiceTest` буде створено в директорії /src/test/java/<ваше ім'я пакета>/services/. Якщо ви не використовували діалогове вікно IDE для створення тесту, просто створіть цей файл самостійно (ліст. 2.6).

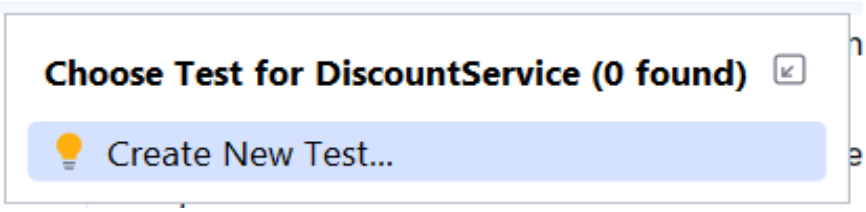

Рисунок 2.1 – Контекстне меню створення для unit-тестів

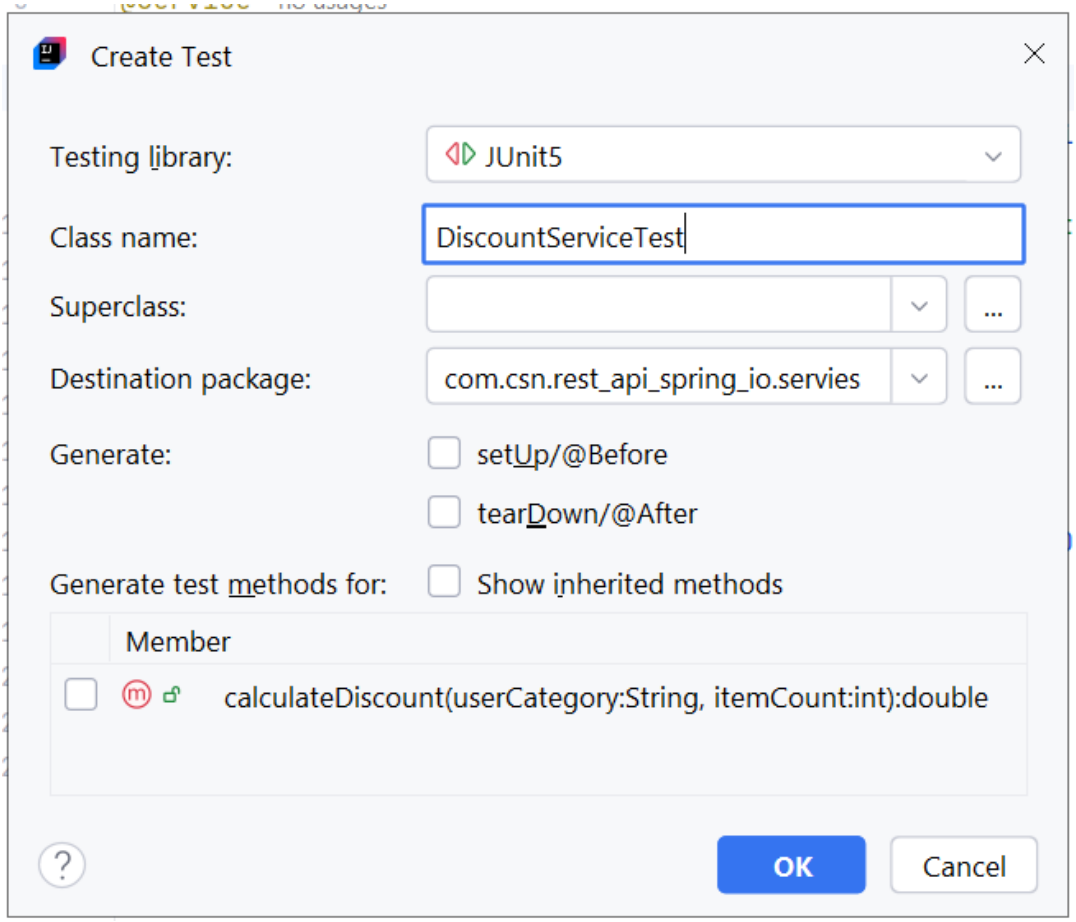

Рисунок 2.2 – Вікно налаштувань JUnit тестів

Лістинг 2.6 – Приклад класу з unit-тестами для сервісу `DiscountService`

```
package com.csn.rest_api_spring_io.servies;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;
/**
 * Тести для DiscountService.
 */
public class DiscountServiceTest {
    private final DiscountService discountService = new
DiscountService();
    @Test
    void testCalculateDiscountForVipUser() {
        double discount =
discountService.calculateDiscount("VIP", 10);
        assertEquals(2.0, discount, "VIP user should get 20%
discount");
    }
    @Test
    void testCalculateDiscountForRegularUser() {
        double discount =
discountService.calculateDiscount("regular", 6);
        assertEquals(0.6, discount, 1e-6, "Regular user
should get 10% discount for more than 5 items");
    }
    @Test
    void testCalculateDiscountForUnknownUser() {
        double discount =
discountService.calculateDiscount("guest", 5);
        assertEquals(0.0, discount, "Guest users should not
get a discount");
    }
```

Кінець лістингу 2.6

```
    @Test
    void testCalculateDiscountWithZeroItems() {
        assertThrows(IllegalArgumentException.class, () ->
discountService.calculateDiscount("VIP", 0), "Should throw
exception when item count is zero or less" );
    }
}
```

При виконанні тести повинні виконуватися правильно (рис. 2.3).

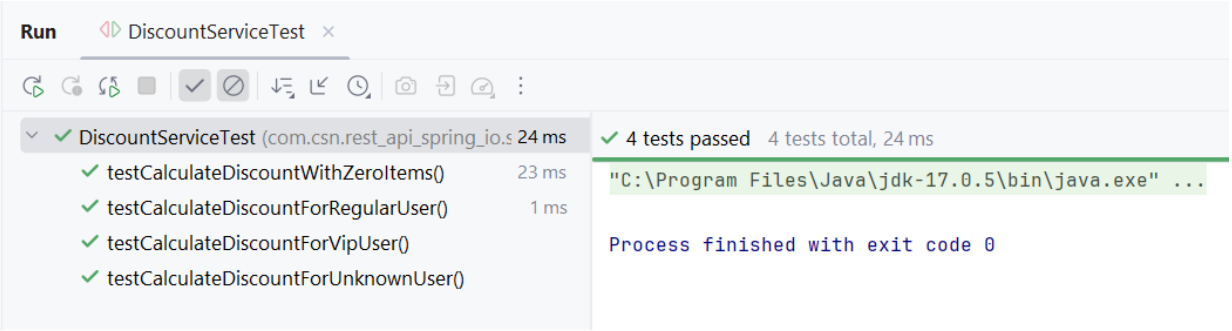

Рисунок 2.3 – Результати успішного виконання тестів

Обговорення тестових сценаріїв. Позитивні випадки: тести перевіряють коректність обчислення знижки для користувачів категорій «VIP» та «постійний». Негативні випадки: включають перевірку для категорії «гість» та ситуації з нульовою кількістю товарів. Обробка винятків: необхідно впевнитися, що для некоректних параметрів генерується виняток IllegalArgumentException.

Крок 2. Додавання імітації взаємодії з Mock (Mockito). Припустимо, що `DiscountService` тепер залежить від іншого сервісу, наприклад, `UserRepository`, який надає дані про користувача. Для тестування можна скористатися Mockito, створивши макет цієї залежності. Створимо новий клас `DiscountServiceWithRepository` із впровадженням залежності (ліст. 2.7). Щоб код працював, створіть модуль репозиторію та клас `UserRepository` у

ньому (ліст. 2.8). Для тестування створіть тест, що відповідає новому класу, за аналогією з попереднім тестом (ліст. 2.9).

Лістинг 2.7 – Код класу `DiscountServiceWithRepository`

```
package com.csn.rest_api_spring_io.servies;
import com.csn.rest_api_spring_io.repository.UserRepository;
import org.springframework.stereotype.Service;
@Service
public class DiscountServiceWithRepository {
    private final UserRepository userRepository;
    public DiscountServiceWithRepository(UserRepository
userRepository) {
        this.userRepository = userRepository;
    }
    public double calculateDiscount(String userId, int
itemCount) {
        if (itemCount <= 0) {
            throw new IllegalArgumentException("Item count
must be greater than 0");
        }
        String userCategory =
userRepository.getUserCategory(userId);
        switch (userCategory.toLowerCase()) {
            case "vip":
                return itemCount * 0.2;
            case "regular":
                return itemCount > 5 ? itemCount * 0.1 : 0.0;
            default:
                return 0.0;
        }
    }
}
```

Лістинг 2.8 – Код класу `UserRepository`

```
package com.csn.rest_api_spring_io.repository;
import java.util.Arrays;
import java.util.List;
import java.util.Random;
@Repository
public class UserRepository {
    private final List<String> categories =
Arrays.asList("vip","regular","others");
    public String getUserCategory(String userId){
    // Для прикладу повертаємо випадкове значення категорії
        Random random = new Random();
        return
categories.get(random.nextInt(categories.size()));
    }
}
```

Лістинг 2.9 – Код класу `DiscountServiceWithRepositoryTest`

```
package com.csn.rest_api_spring_io.servies;
import com.csn.rest_api_spring_io.repository.UserRepository;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;
class DiscountServiceWithRepositoryTest {
    private UserRepository userRepository;
    private DiscountServiceWithRepository discountService;
    @BeforeEach
    void setUp() {
        userRepository = mock(UserRepository.class);
        discountService = new
DiscountServiceWithRepository(userRepository);
    }
```

Кінець лістингу 2.9

```
    @Test
    void testCalculateDiscountWithMockedUser() {

when(userRepository.getUserCategory("123")).thenReturn("VIP");
        double discount =
discountService.calculateDiscount("123", 10);
        assertEquals(2.0, discount);
        verify(userRepository,
times(1)).getUserCategory("123");
    }
    @Test
    void testCalculateDiscountForUnknownUser() {

when(userRepository.getUserCategory("456")).thenReturn("guest");
        double discount =
discountService.calculateDiscount("456", 5);
        assertEquals(0.0, discount);
        verify(userRepository,
times(1)).getUserCategory(anyString());
    }
}
```

Тести повинні виконуватися без помилок (рис. 2.4). Що потрібно врахувати при створенні останнього тестового класу з використанням об’єктів mock:

– створення макета: необхідно створити `mock` для `UserRepository` – фальшивої реалізації, яка не взаємодіє з реальною базою даних;

– заглушки методів: за допомогою `when().thenReturn()` задаються значення, які мають повертатися під час виклику методів макета;

– перевірка викликів: `verify()` дозволяє переконатися, що методи макета були викликані з очікуваними параметрами.

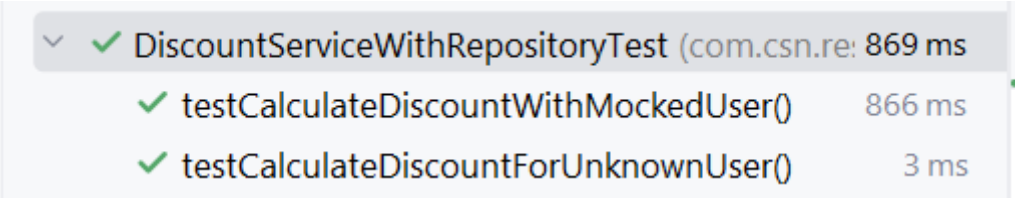

Рисунок 2.4 - Результати успішного виконання тестів

Основні помилки при роботі з Mockito та способи їх усунення:

– відсутність заглушок: якщо забути застосувати `when().thenReturn()`, макет за замовчуванням повертатиме `null`;

– ігнорування перевірки викликів: без `verify()` неможливо переконатися, що методи тестованого класу правильно взаємодіють із макетами;

– необроблені винятки: пропуск перевірки винятків може призвести до того, що тести не виявлять критичних помилок у коді.

2.2.4 Інтеграційне тестування мікросервісів за допомогою MockMvc

Уявімо мікросервіс, який відповідає за обробку замовлень в онлайн-магазині. Він приймає HTTP-запити, звертається до сервісів для перевірки наявності товарів на складі та взаємодіє з базою даних для збереження інформації про замовлення. Це комплекс взаємопов’язаних компонентів. Навіть якщо ви протестували їх окремо в модульних тестах, це не гарантує, що система працює правильно в цілому. Саме для цього застосовується інтеграційне тестування.

Інтеграційне тестування на рівні мікросервісу дозволяє:

– перевіряти взаємодію між контролерами, сервісами, репозиторіями та іншими компонентами;

– переконатися, що мікросервіс функціонує коректно, без потреби у реальних зовнішніх залежностях;

– емулювати справжні HTTP-запити та оцінювати обробку запитів та формування відповідей.

Основний інструмент для інтеграційного тестування MockMvc – це зручний засіб із Spring Test, який дозволяє:

– імітувати HTTP-запити до програми без підняття реального вебсервера;

– перевіряти, як контролери обробляють запити, повертають відповіді та взаємодіють із сервісами;

– моделювати реальні сценарії API, включно з параметрами запиту, тілом повідомлення та налаштуваннями заголовків.

Завдяки MockMvc можна тестувати веб-рівень додатку ізольовано, без необхідності запускати весь сервер.

MockMvc забезпечує зручний та ефективний спосіб перевірки веб-рівня програми:

– швидкість виконання: тести з MockMvc проходять швидше, ніж повноцінні інтеграційні тести з підняттям серверу, що значно економить час, особливо у великих проєктах;

– ізоляція тестованого шару: якщо база даних або зовнішні сервіси ще недоступні, це не завадить тестуванню, оскільки їх можна змокати;

– гнучкий API: MockMvc дозволяє легко формувати запити та перевіряти, як контролери обробляють їх і формують відповіді.

Для налаштування MockMvc у тестовому класі використовують такі анотації:

– `@SpringBootTest` – піднімає контекст Spring для тестування;

– `@AutoConfigureMockMvc` – автоматично конфігурує MockMvc для роботи з тестами.

Виконайте тестування REST API за допомогою MockMvc.

Припустимо, у нас є мікросервіс для управління книгами, з REST API для роботи з колекцією книг. Створіть модуль контролера у своєму проєкті та додайте до нього клас BookService (ліст. 2.10). У нас є два методи: GET /books/{id}, який повертає книгу за її ID, та POST /books, який додає нову книгу.

Лістинг 2.10 – Код класу сервісу `BookService`

```
@RestController
@RequestMapping("/books")
public class BookController {
    private final BookService bookService;
    public BookController(BookService bookService) {
        this.bookService = bookService;
    }
    @GetMapping("/{id}")
    public ResponseEntity<Book> getBook(@PathVariable Integer
id) {
        return
ResponseEntity.ok(bookService.findBookById(id));
    }
    @PostMapping
    public ResponseEntity<Book> createBook(@RequestBody Book
book) {
        return
ResponseEntity.status(HttpStatus.CREATED).body(bookService.cr
eate(book));
    }
}
```

Зверніть увагу на використання класу `ResponseEntity<T>`. У Spring Boot `ResponseEntity<T>` виступає універсальною обгорткою для HTTP-відповіді. Він дозволяє керувати не лише самим тілом відповіді, а й іншими аспектами:

– HTTP-статус: наприклад, 200 OK, 404 Not Found, 201 Created;

– HTTP-заголовки: Content-Type, Location або власні заголовки;

– тіло відповіді: об’єкт, JSON, рядок тощо.

Без застосування `ResponseEntity` контролер зазвичай повертає лише дані (наприклад, об’єкт), а Spring автоматично встановлює статус 200 OK. Проте у

REST API часто потрібно гнучко керувати кодами відповіді та заголовками – саме для цього і використовується `ResponseEntity`.

Коли застосовувати `ResponseEntity`:

– якщо потрібно повернути не лише дані, а й конкретний HTTP-код;

– коли необхідно додати власні заголовки до відповіді;

– якщо REST-стиль вимагає повідомити, що ресурс не знайдено (404), створено (201) або видалено (204).

Тепер напишемо тести для цього контролера з MockMvc. Для початку створимо тестовий клас (ліст. 2.11).

Лістинг 2.11 – Код класу `BookControllerTest` із unit-тестами

```
package com.csn.rest_api_spring_io.controller;
import com.csn.rest_api_spring_io.model.Book;
import com.csn.rest_api_spring_io.servies.BookService;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import
org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.AutoC
onfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.*;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatc
hers.*;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBu
ilders.*;
```

Продовження лістингу 2.11

```
@SpringBootTest
@AutoConfigureMockMvc
class BookControllerTest {
    @Autowired
    private MockMvc mockMvc;
    @MockBean
    private BookService bookService; // Мокуємо Відправляємо
залежність сервісу
    @Test
    void testGetBook() throws Exception {
        // Дані для тесту
        Book book = new Book(1, "The Hobbit", "J.R.R.
Tolkien");
Mockito.when(bookService.findBookById(1)).thenReturn(book);
    // Відправляємо GET запит і перевіряєм результат
    mockMvc.perform(get("/books/1"))
        .andExpect(status().isOk()) // Очікуємо статус 200 OK
        .andExpect(jsonPath("$.title").value("The Hobbit"))
// Перевіряємо поле title
        .andExpect(jsonPath("$.author").value("J.R.R.
Tolkien")); // Перевіряємо поле author
    }
    @Test
    void testCreateBook() throws Exception {
        // Дані для тесту
        Book book = new Book(null, "1984", "George Orwell");
        Book savedBook = new Book(2, "1984", "George
Orwell");
Mockito.when(bookService.create(Mockito.any())).thenReturn(sa
vedBook);
        // Відправляємо POST запит із JSON-даними
        String bookJson = """
```

Кінець лістингу 2.11

```
            {
             "title": "1984",
             "author": "George Orwell"
            }
             """;
        mockMvc.perform(post("/books")
             .contentType(MediaType.APPLICATION_JSON)
             .content(bookJson))
             .andExpect(status().isCreated()) // Очікуємо
статус 201 Created
             .andExpect(jsonPath("$.id").value(2L)) //
Перевіряємо, що ID книги = 2
             .andExpect(jsonPath("$.title").value("1984")) //
Перевіряємо назву
             .andExpect(jsonPath("$.author").value("George
Orwell")); // Перевіряємо автора
    }
}
```

Що відбувається у цьому прикладі. Анотації:

– `@SpringBootTest` піднімає повний контекст Spring для тестування;

– `@AutoConfigureMockMvc` автоматично налаштовує MockMvc для відправки запитів до контролерів;

– `@MockBean` створює mock об’єкт із використання фреймворку для BookService, що дозволяє ізолювати тестований компонент від реальної залежності.

Випробування:

– у `testGetBook` ми створювали метод-заглушку для методу `findBookById` сервісу, щоб він повертав об’єкт `Book`. Потім відправляємо `GET`-запит і перевіряємо, що відповідь містить правильні дані;

– у `testCreateBook` створюємо метод-заглушку для методу `saveBook`, надсилаємо `POST`-запит із JSON-даними та перевіряємо, що відповідь повертає дані створеної книги.

API MockMvc:

– `mockMvc.perform(...)` виконує HTTP-запит;

– `andExpect(...)` використовується для перевірки результатів – від HTTP-статусу до вмісту JSON у відповіді.

При використанні MockMvc варто звернути увагу на:

– залежність MockMvc або `@AutoConfigureMockMvc` не була оголошена, і в цьому випадку тести не зможуть використовувати MockMvc;

– помилки серіалізації у форматі JSON. Перевірте, чи підключена бібліотека Jackson до вашого проєкту;

– відсутність mocks для залежностей. MockMvc лише тестує контролер, тому обов'язково замокає служби через `@MockBean`.

MockMvc – потужний інструмент для тестування взаємодії з вашими контролерами, що дозволяє тестувати HTTP-запити та швидко виявляти помилки, які могли б з’явитися лише в реальному світі. З його допомогою ви можете бути впевнені в коректній роботі вашого API ще до розгортання.

2.3 Завдання до лабораторної роботи

У практичній частині свідомо пропущено реалізацію окремих елементів. Модельні класи книги з полями `id`, `title`, `author`. Сервіс `BookService`, який має містити щонайменше два методи: `create` та `findBookById`. Ці компоненти потрібно реалізувати самостійно так, щоб вони були сумісні з усім іншим кодом із практичного завдання.

У попередньому практичному завданні було реалізовано REST API. У межах цієї лабораторної роботи необхідно забезпечити його перевірку за допомогою тестування. Зокрема, потрібно розробити як модульні тести, що

перевіряють роботу окремих компонентів у ізоляції, так і інтеграційні тести, які охоплюють взаємодію між різними шарами застосунку та базою даних. Такий підхід дозволить гарантувати стабільність і коректність функціонування API на всіх рівнях системи.

Тести CRUD: варто створити тести, що перевіряють операції створення, читання, оновлення та видалення. Не обов'язково тестувати абсолютно всі методи кожного класу – достатньо показати приклади для кожного типу тестування.

Для модульних тестів потрібно продемонструвати:

– обробку некоректних даних: сервіс повинен викидати виняток або повертати помилку;

– перевірку роботи з «порожніми» даними (наприклад, пустий список чи `null`);

– тестування з максимально можливими та мінімальними вхідними параметрами.

Для інтеграційних тестів варто показати:

– як відбувається взаємодія між контролерами, сервісами, репозиторіями та іншими складовими;

– переконатися, що мікросервіс функціонує правильно без підключення до реальних зовнішніх систем;

– імітацію реальних HTTP-запитів і перевірку, як застосунок формує відповіді на них.

2.4 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.

2.5 Контрольні питання

1. Які основні підходи існують для тестування REST API у Spring Boot і чим відрізняються юніт-тести від інтеграційних тестів?

2. Як анотація `@SpringBootTest` допомагає у створенні інтеграційних тестів для REST API?

3. Яку роль відіграє бібліотека Mockito при тестуванні контролерів і сервісів у Spring Boot?

4. Як за допомогою `@MockBean` можна замінити залежності реальних сервісів у тестах?

5. Для чого використовується MockMvc і які можливості воно надає для тестування REST-контролерів?

6. Як за допомогою MockMvc перевірити статус відповіді HTTP та вміст JSON-об’єкта?

7. Чим відрізняються анотації `@WebMvcTest` і `@SpringBootTest`, і коли доцільно використовувати кожну з них?

8. Як можна протестувати обробку винятків у REST API за допомогою MockMvc?

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.

8. Mastering Spring Boot 3.0: From Basics to Advanced – Implement Architectural Patterns and Advanced Testing Strategies / Meric, A. // Packt Publishing. 2024 – 256 p.

9. Learning Spring Boot 3.0 – Third Edition: Simplify the Development of Production-Grade Applications Using Java and Spring. / Turnquist, G. L. // Packt Publishing. 2022 – 270 p.