Міністерство освіти і науки України
Національний університет «Одеська політехніка»

Інститут комп’ютерних систем

Кафедра інформаційних систем

Ахтеменко Денис Вікторович,

студент групи АІ-224

КУРСОВА РОБОТА

Банківська система на базі Java Spring

Спеціальність:

122 Комп’ютерні науки

Освітня програма: Комп’ютерні науки

Керівник:

Годовиченко Микола Анатолійович,

Канд. техн. наук

Одеса – 2024

ЗМІСТ

Національний університет «Одеська політехніка»

Інститут комп’ютерних систем
Кафедра інформаційних систем

ЗАВДАННЯ
НА КУРСОВУ РОБОТУ

студенту _Ахтеменку Денису Вікторовичу група АІ-224___

1. Тема роботи_«Розробити банківську систему на базі Java Spring_____

2. Термін здачі студентом закінченої роботи __05.06.2024__

3. Початкові дані до проекту (роботи)_Варіант 2: Впровадження журналювання за допомогою Spring AOP, Розробити відповідні аспекти для впровадження журналювання подій в шарі REST-контролерів, сервісів та репозиторіїв. Під'єднати та налаштувати процес журналювання подій за допомогою SLF4J та Logback. Реалізація REST API для проведення фінансових операцій. Використання Spring AOP для моніторингу безпеки транзакцій. Інтеграція з СУБД для зберігання даних користувачів та транзакцій

4. Зміст розрахунково-пояснювальної записки (перелік питань, які належить розробити)_постановка задачі, проектування бази даних, вибір програмного забезпечення, створення бази даних, маніпулювання даними, створення користувачів і призначення прав доступу

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
Знімки екрану з середовища розробки_____

Завдання видано 4.04.2024 _____
(підпис викладача)

Завдання прийнято до виконання 4.04.2024 _____
(підпис студента)

АНОТАЦІЇ

У цій роботі розглядається процес створення банківської системи на основі Java Spring. Основна увага приділяється організації веб-шару для обробки REST-запитів, розробці бізнес-логіки додатку та роботі з базою даних. Описано архітектуру, яка допомагає ефективно керувати банківськими операціями та взаємодіяти між різними компонентами системи. Також детально розглянуто автоматизацію рутинних процесів, валідацію даних і забезпечення безпеки додатку. Проведено аналіз продуктивності системи після впровадження та визначено шляхи для її подальшого вдосконалення.

ABSTRACT

This paper describes the process of creating a banking system based on Java Spring. The main attention is paid to the organisation of the web layer for processing REST requests, development of the application business logic, and work with the database. The architecture is described, which helps to effectively manage banking operations and interact between different components of the system. Automation of routine processes, data validation, and application security are also discussed in detail. The article analyses the system performance after implementation and identifies ways to further improve it.

ВСТУП

Оскільки цифрові технології активно розвиваються, а автоматизація процесів стає необхідною для успішного функціонування багатьох сфер бізнесу, впровадження банківських систем на основі Java Spring набуває все більшої актуальності. У сучасних умовах швидкого зростання банківського сектора та збільшення обсягу транзакцій, ефективне управління даними та забезпечення безпеки є критичними для стабільної роботи банків. Ця робота спрямована на створення банківської системи, яка дозволяє автоматизувати ключові процеси, забезпечити високу продуктивність та безпеку даних.

Банківські системи на базі Java Spring дозволяють організувати обробку запитів клієнтів, виконання транзакцій, управління рахунками та забезпечити зручний доступ до фінансових послуг. Це допомагає банкам оперативно реагувати на запити клієнтів, мінімізувати ризики, пов’язані з помилками, та підвищити загальну ефективність обслуговування. Крім того, сучасні рішення з автоматизації полегшують управління великими обсягами інформації, забезпечуючи швидке реагування на зміни ринку та вимоги клієнтів.

Важливо зазначити, що подібні системи є не лише інструментом для вирішення поточних завдань, але й мають стратегічне значення. Вони дозволяють банкам залишатися гнучкими, швидко адаптуватися до змін, впроваджувати нові продукти та послуги, що в свою чергу сприяє зростанню бізнесу та зміцненню конкурентоспроможності.

Таким чином, розробка банківських систем на базі Java Spring є важливим та актуальним завданням, яке відповідає сучасним тенденціям ринку та потребам банківського бізнесу.

1 ПРОЕКТУВАННЯ І РОЗРОБКА ВЕБ-ШАРУ

1.1 Вибір технологій

Для розробки веб-шару додатку було обрано **Spring Boot**, і це рішення обґрунтовується декількома важливими перевагами:

1. **Простота налаштування та швидкий старт**. Spring Boot значно спрощує створення веб-додатків, пропонуючи готові шаблони для типових завдань, що дозволяє уникнути великого обсягу конфігураційного коду. Для побудови REST-сервісів Spring Boot є оптимальним вибором, оскільки він надає мінімальні вимоги до конфігурації та дозволяє розпочати роботу майже одразу.
2. **Потужна підтримка REST API**. Spring Boot інтегрований із Spring MVC, що надає готові інструменти для створення RESTful сервісів. Завдяки цьому фреймворку легко організувати роботу з HTTP-запитами (GET, POST, PUT, DELETE), налаштувати маршрутизацію запитів та автоматизувати процес конвертації даних між форматами JSON та Java-об'єктами.
3. **Масштабованість та модульність**. Spring Boot дозволяє створювати як невеликі додатки, так і масштабовані системи. Завдяки модульній архітектурі фреймворку можна додавати нові функції та інтегрувати різні технології без суттєвих змін у базовій архітектурі.
4. **Підтримка інверсії управління (IoC) та ін'єкції залежностей (DI)**. Це спрощує роботу з різними сервісами та модулями, дозволяючи автоматизувати управління залежностями між компонентами додатку, що робить систему гнучкою та легко розширюваною.
5. **Готова підтримка безпеки**. Важливим аспектом для банківських додатків є безпека. Spring Boot має вбудовані інструменти для інтеграції

безпеки, зокрема Spring Security, який можна налаштувати для захисту API за допомогою різних механізмів аутентифікації та авторизації.

В результаті вибір Spring Boot був оптимальним для створення веб-шару банківської системи, оскільки він дозволив поєднати швидкість розробки, гнучкість і надійність, що особливо важливо для веб-додатків із високими вимогами до продуктивності та безпеки.

1.2 Створення контролерів

Контролери у Spring Boot є ключовим елементом, що забезпечує взаємодію між користувачем і сервером. У банківській системі контролери були створені для обробки різних типів HTTP-запитів, таких як **GET, POST, PUT** та **DELETE**. Кожен тип запиту відповідає певній операції з даними: отримання, створення, оновлення та видалення ресурсів (наприклад, користувачів, транзакцій, рахунків).

Створення REST-контролерів для обробки запитів

У Spring Boot контролери налаштовуються за допомогою спеціальних анотацій, що значно спрощує процес маршрутизації запитів. Наприклад, анотація @RestController вказує, що цей клас буде обробляти HTTP-запити і повертати результати у форматі JSON або XML. Усі запити до конкретного контролера спрямовуються на базі анотації @RequestMapping, яка визначає загальний шлях для запитів.

Приклад контролера для роботи з користувачами:

```
@RestController
@RequestMapping("/api/users")
public class UserController {
```

```
    @Autowired
    private UserService userService;

    @GetMapping
    public ResponseEntity<List<User>> getAllUsers() {
        return ResponseEntity.ok(userService.getAllUsers());
    }

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable Long id)
{
        User user = userService.getUserById(id);
        if (user != null) {
            return ResponseEntity.ok(user);
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    @PostMapping
    public ResponseEntity<User> createUser(@RequestBody User user)
{
        User createdUser = userService.createUser(user);
        return ResponseEntity.ok(createdUser);
    }

    @PutMapping("/{id}")
    public ResponseEntity<User> updateUser(@PathVariable Long id,
@RequestBody User user) {
        User updatedUser = userService.updateUser(id, user);
        if (updatedUser != null) {
            return ResponseEntity.ok(updatedUser);
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<User> deleteUser(@PathVariable Long id) {
        User deletedUser = userService.deleteUser(id);
        if (deletedUser != null) {
            return ResponseEntity.ok(deletedUser);
        } else {
            return ResponseEntity.notFound().build();
        }
    }
}
```

Огляд методів:

1. **getAllUsers()** — Метод обробляє запит GET /api/users і повертає список усіх користувачів у форматі JSON. Для обробки цього запиту використовується анотація @GetMapping.
2. **getUserById()** — Метод обробляє запит GET /api/users/{id}, отримуючи користувача за його ID. Анотація @PathVariable вказує, що параметр id буде передаватися в URL-запиті. Якщо користувач існує, сервер повертає об'єкт User і статус 200 OK. Якщо користувач не знайдений, повертається статус 404 Not Found.
3. **createUser()** — Цей метод обробляє запит на створення нового користувача за допомогою HTTP методу POST. Анотація @RequestBody використовується для передачі даних у форматі JSON, які автоматично конвертуються у Java-об'єкт.
4. **updateUser()** — Метод обробляє запит PUT /api/users/{id} для оновлення даних користувача. Так само, як і при створенні, дані передаються у тілі запиту за допомогою анотації @RequestBody.
5. **deleteUser()** — Для видалення користувача обробляється запит DELETE /api/users/{id}. Якщо користувача успішно видалено, сервер повертає статус 200 OK. Якщо користувача не знайдено, повертається статус 404 Not Found.

Переваги використання REST-контролерів у Spring Boot:

- Простота у створенні та налаштуванні завдяки використанню анотацій.
- Автоматична обробка JSON даних.
- Підтримка різних типів HTTP-запитів, що забезпечує повноцінну роботу з ресурсами додатку.

- Гнучка обробка запитів та відповідей, включаючи повернення коректних статусів HTTP.

1.3 Обробка запитів та відповідей

У Spring Boot обробка запитів і відповідей базується на концепції об'єктів, які передаються між клієнтом і сервером у вигляді HTTP-запитів. Використання спеціальних об'єктів для передачі даних спрощує процес обробки запитів, а об'єкт ResponseEntity дозволяє керувати не тільки даними, що повертаються, але і статусом відповіді.

Обробка запитів

При отриманні HTTP-запиту сервер перетворює вхідні дані (як правило, у форматі JSON) у Java-об'єкти, які контролери можуть обробляти. Цей процес автоматично реалізується завдяки використанню анотації **@RequestBody**.

Приклад обробки POST-запиту для створення користувача:

```
@PostMapping
public ResponseEntity<User> createUser(@RequestBody User user) {
    User createdUser = userService.createUser(user);
    return ResponseEntity.ok(createdUser);
}
```

У цьому випадку JSON-дані, надіслані клієнтом, автоматично перетворюються у Java-об'єкт класу User за допомогою Jackson (вбудований механізм Spring Boot для серіалізації/десеріалізації).

Обробка відповідей

Для повернення результату використовується об'єкт **ResponseEntity**, який дозволяє не тільки передавати дані, але й керувати HTTP-статусами.

Приклад використання ResponseEntity:

```
@GetMapping("/{id}")
public ResponseEntity<User> getUserById(@PathVariable Long id) {
    User user = userService.getUserById(id);
    if (user != null) {
        return ResponseEntity.ok(user); // Повертається користувач і статус 200 OK
    } else {
        return ResponseEntity.notFound().build(); // Повертається статус 404 Not Found
    }
}
```

У прикладі вище:

- Якщо користувач знайдений, повертається об'єкт User разом зі статусом 200 OK.
- Якщо користувача не знайдено, сервер повертає статус 404 Not Found, що дозволяє клієнту розуміти, що ресурс відсутній.

Таблиця 1 – Перелік можливих HTTP-статусів при обробці запитів

Значення HTTP-статусу	Опис статуса
200 OK	запит успішно оброблено
201 Created	ресурс успішно створено
400 Bad Request	запит некоректний, наприклад, через помилкові дані
404 Not Found	запитуваний ресурс не знайдено
500 Internal Server Error	сервер зіткнувся з несподіваною помилкою

Нижче показан приклад виконання запиту **GET** для моделі User в Postman на рисунку 1. Так як маршрутизація запитів вказана вірно, тому операція буде успішною **200 ОК.**

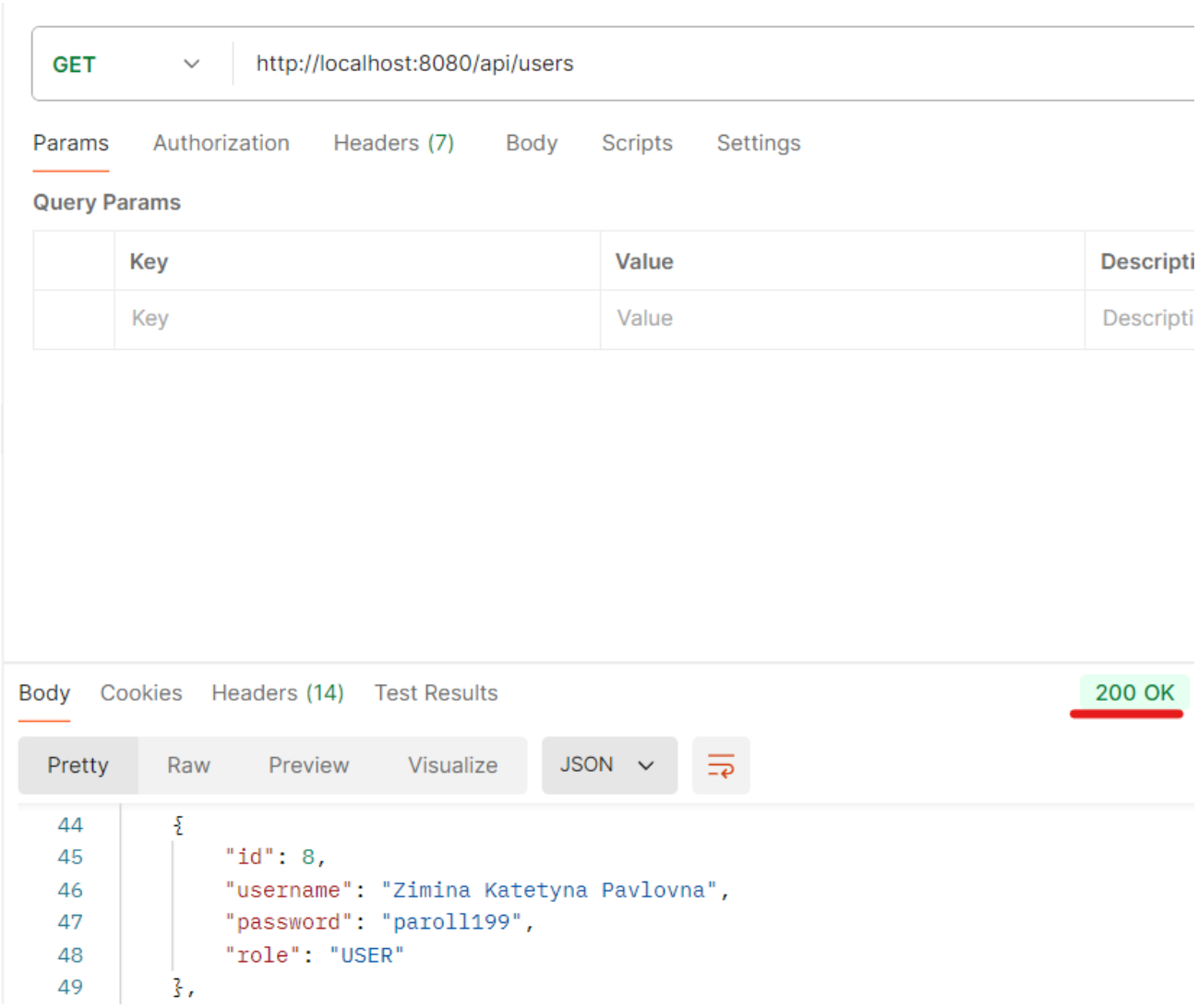

Рис.1 – приклад виконання запиту GET на 200 ОК в Postman

Для наступної перевірки обробки даних використаємо помилковий маршрут до даних. Замість правильного маршруту до методу виведення всіх користувачів:

```
http://localhost:8080/api/users
```

Будемо використовувати помилковий маршрут:

```
http://localhost:8080/api/user
```

Так як такого шляху немає, в результаті будемо отримувати результат **404 Not Found**, який означає що такого ресурсу немає. На рисунку 2 продемонстровано виконання такого запиту в Postman.

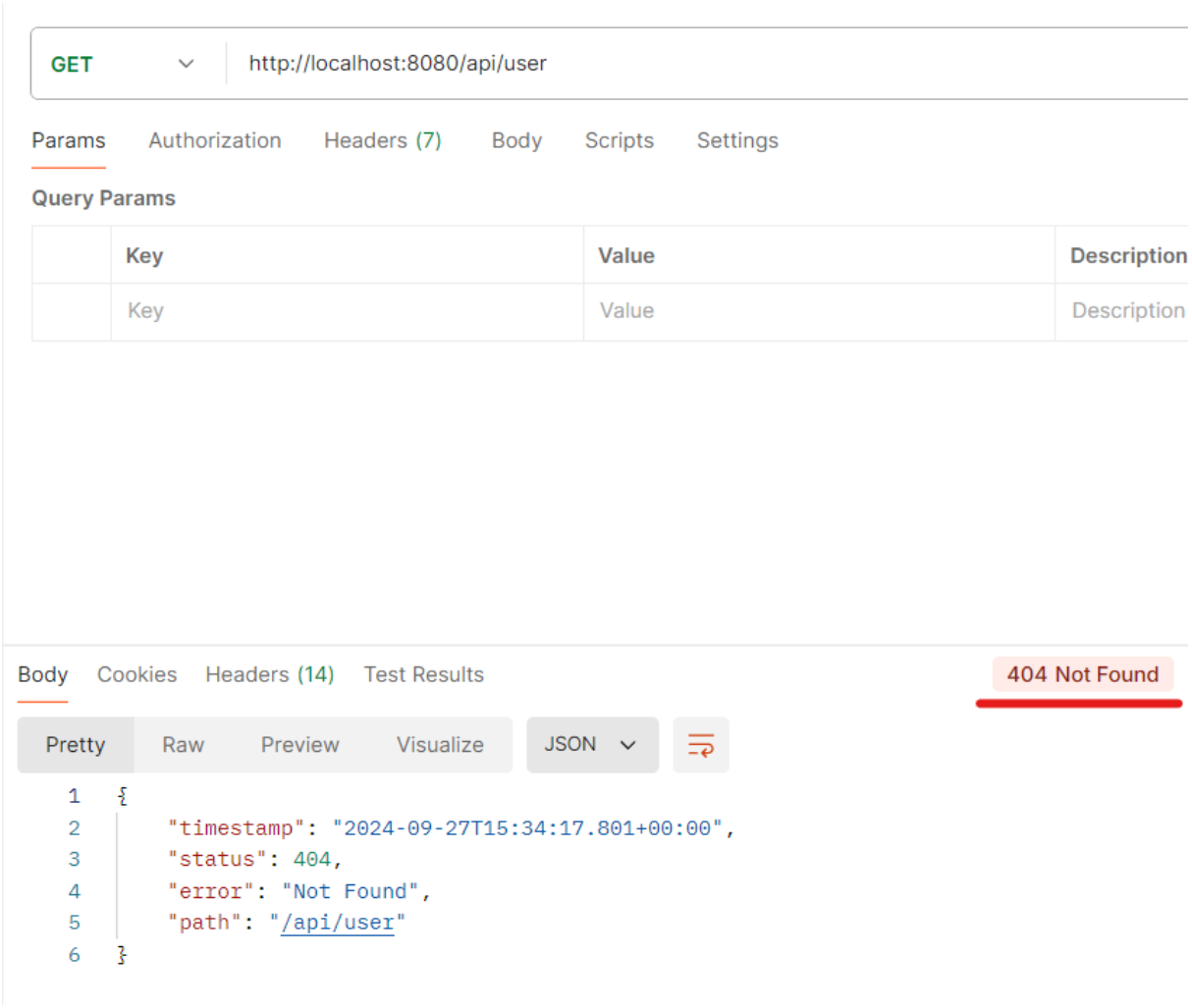

Рис. 2 – Приклад виконання помилкового запиту на відповідь 404 в Postman

1.4 Документація API

Документування API є важливим кроком для будь-якого веб-додатку, оскільки це дозволяє іншим розробникам і тестувальникам зрозуміти, як взаємодіяти з додатком, які запити підтримуються, які дані очікуються та як обробляються відповіді.

Використання Postman

Postman — це потужний інструмент для тестування, документування та автоматизації запитів до API. Під час розробки банківської системи Postman використовувався для:

1. **Тестування API**. З його допомогою можна легко відправляти запити на різні ендпоінти додатку і перевіряти їхню роботу. Postman підтримує всі типи HTTP-запитів (GET, POST, PUT, DELETE) і дозволяє передавати параметри запитів, заголовки та тіла запитів.

   Наприклад, для перевірки роботи ендпоінту для створення користувача (POST-запит до /api/users), в Postman можна надіслати тіло запиту у форматі JSON:

```
{
    "name": "Denis Akhtemenko",
    "email": "den.akhtem21@gmail.com"
}
```

   Після відправлення запиту Postman відобразить відповідь сервера у вигляді JSON-об'єкта, що містить новоствореного користувача:

```
{
    "id": 1,
    "name": "Denis Akhtemenko",
    "email": "den.akhtem21@gmail.com",
    "balance": 1500.50
```

```
}
```

2. **Колекції запитів**. Postman дозволяє створювати колекції запитів, що полегшує організацію тестування. Наприклад, для банківської системи можна створити колекції для управління користувачами, транзакціями або рахунками. Це дає змогу зберігати і повторно використовувати всі необхідні запити під час розробки.
3. **Документування API**. Postman також може бути використаний для генерації документації. Ви можете зберегти всі запити та параметри, а потім згенерувати автоматичну документацію, яку можна поділитися з іншими розробниками або клієнтами.

2 ПРОЕКТУВАННЯ І РОЗРОБКА СЕРВІСНОГО ШАРУ

2.1 Структура сервісного шару

Сервісний шар є ключовим елементом архітектури, який відповідає за реалізацію бізнес-логіки додатку та забезпечує взаємодію між веб-шаром і шаром доступу до даних (репозиторії). У цьому проекті сервісний шар організований у вигляді окремих класів-сервісів, кожен з яких відповідає за конкретну бізнес-діяльність.

Основна ідея сервісного шару полягає у відокремленні бізнес-логіки від контролерів, що відповідають за обробку HTTP-запитів, і від репозиторіїв, що керують доступом до бази даних. Це дозволяє дотримуватися принципу розділення обов'язків та робить код більш модульним і легко підтримуваним.

У цьому проекті було створено декілька класів-сервісів, кожен з яких відповідає за певну бізнес-логіку:

- **UserService** — керує операціями з користувачами (створення, отримання, оновлення, видалення користувачів).
- **TransactionService** — відповідальний за обробку фінансових операцій.
- **AccountService** — працює з рахунками користувачів (отримання залишку, відкриття або закриття рахунку).
- **CardService** — забезпечує управління картками користувачів.
- **FinancialProductService** — відповідає за управління фінансовими продуктами (кредити, депозити тощо).

Кожен сервіс взаємодіє з відповідним репозиторієм для доступу до даних. Наприклад, клас **UserService** взаємодіє з класом **UserRepository**, щоб виконувати операції над сутністю користувача.

Взаємодія між контролерами і сервісами

Для того щоб забезпечити взаємодію між контролерами (веб-шаром) і сервісами, використовується анотація **@Autowired**, яка дозволяє автоматично інжектувати залежності. Це дозволяє контролерам викликати методи сервісів для обробки запитів.

Приклад інжекції залежності у контролері:

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping
    public ResponseEntity<List<User>> getAllUsers() {
        return ResponseEntity.ok(userService.getAllUsers());
    }

}
```

У цьому прикладі контролер **UserController** використовує сервіс **UserService** для отримання списку користувачів. Це дозволяє контролеру зосередитися виключно на обробці HTTP-запитів, тоді як усі операції з даними виконує сервіс.

2.2 Бізнес-логіка

Бізнес-логіка додатку реалізована в кожному сервісі окремо, і кожен сервіс виконує специфічні операції з даними, що надходять із веб-шару. Наприклад, у **UserService** реалізовані методи для створення нового користувача, його оновлення, видалення, а також перевірки унікальності користувача перед його додаванням до системи.

Основна ідея бізнес-логіки полягає в тому, що всі дані, що надходять із контролерів, обробляються на рівні сервісів. Тут відбувається валідація даних, застосування бізнес-правил та підготовка даних перед передачею до шару зберігання.

Приклад простої бізнес-логіки у UserService:

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public User createUser(User user) {
        if (userRepository.existsByEmail(user.getEmail())) {
            throw new IllegalArgumentException("Користувач з такою
електронною поштою вже існує.");
        }
        return userRepository.save(user);
    }

    public User updateUser(Long id, User user) {
        Optional<User> existingUser = userRepository.findById(id);
        if (existingUser.isPresent()) {
            user.setId(id);
            return userRepository.save(user);
        }
        return null;
    }

    public User deleteUser(Long id) {
        Optional<User> existingUser = userRepository.findById(id);
```

```
        if (existingUser.isPresent()) {
            userRepository.deleteById(id);
            return existingUser.get();
        }
        return null;
    }
}
```

У цьому прикладі бізнес-логіка методу **createUser()** перевіряє, чи існує користувач з вказаною електронною поштою у базі даних. Якщо такий користувач вже є, кидається виключення **IllegalArgumentException**. Якщо користувач не знайдений, він додається до бази даних за допомогою методу **save()** репозиторію.

Таблиця 2 – Список використаних методів у прикладі коду UserService

Назва методу	Опис методу
createUser()	перевіряє, чи існує користувач із такою ж електронною адресою, і, якщо його немає, додає до бази.
updateUser()	оновлює дані користувача, якщо він існує в базі даних
deleteUser()	видаляє користувача за його ID

2.3 Операції CRUD

Основні операції CRUD (створення, читання, оновлення, видалення) реалізовані через репозиторії, які забезпечують взаємодію з базою даних. **JpaRepository** автоматично надає методи для виконання цих операцій без необхідності додаткового написання SQL-запитів.

Приклад операції створення користувача:

```
public User createUser(User user) {
    return userRepository.save(user);
```

```
}
```

Приклад операції читання:

```
public List<User> getAllUsers() {
    return userRepository.findAll();
}

public User getUserById(Long id) {
    return userRepository.findById(id).orElse(null);
}
```

Приклад операції оновлення, цей метод спочатку перевіряє, чи існує користувач з таким ID, і якщо так - оновлює його дані:

```
public User updateUser(Long id, User user) {
    Optional<User> existingUser = userRepository.findById(id);
    if (existingUser.isPresent()) {
        user.setId(id);
        return userRepository.save(user);
    }
    return null;
}
```

Приклад операції видалення, метод видаляє користувача з бази даних за його ID:

```
public User deleteUser(Long id) {
    Optional<User> existingUser = userRepository.findById(id);
    if (existingUser.isPresent()) {
        userRepository.deleteById(id);
        return existingUser.get();
    }
    return null;
}
```

Таблиця 3 – Список методів, які були присутні в операціях CRUD

Назви методів	Опис методу
save()	метод репозиторію, який зберігає новий об'єкт у базу даних або оновлює наявний, якщо він вже існує

findAll()	метод для отримання всіх користувачів з бази даних.
findById()	метод для пошуку користувача за його унікальним ID.

Приклади виконання операцій CRUD

Для виконання запиту для виведення даних (GET, READ) одного користувача, потрібно використати такий шлях:

```
http://localhost:8080/api/users/1
```

На рисунку 3 показан результат введення цього шляху

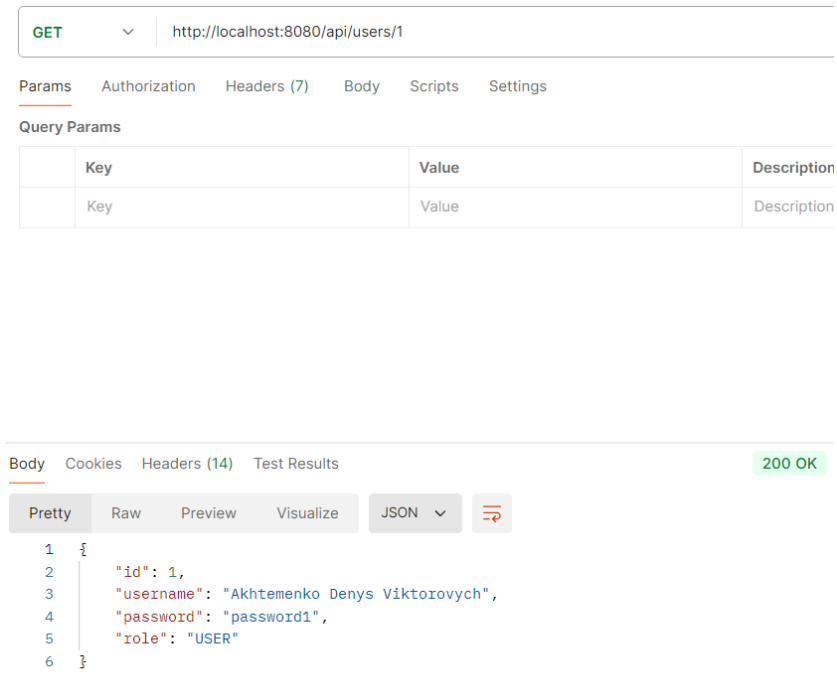

Рис.3 – Приклад отримання даних користувача за id

Для виконання запиту для створення даних (POST, CREATE) користувача, потрібно використати такий шлях:

```
http://localhost:8080/api/users
```

На рисунку 4 показан результат введення цього шляху

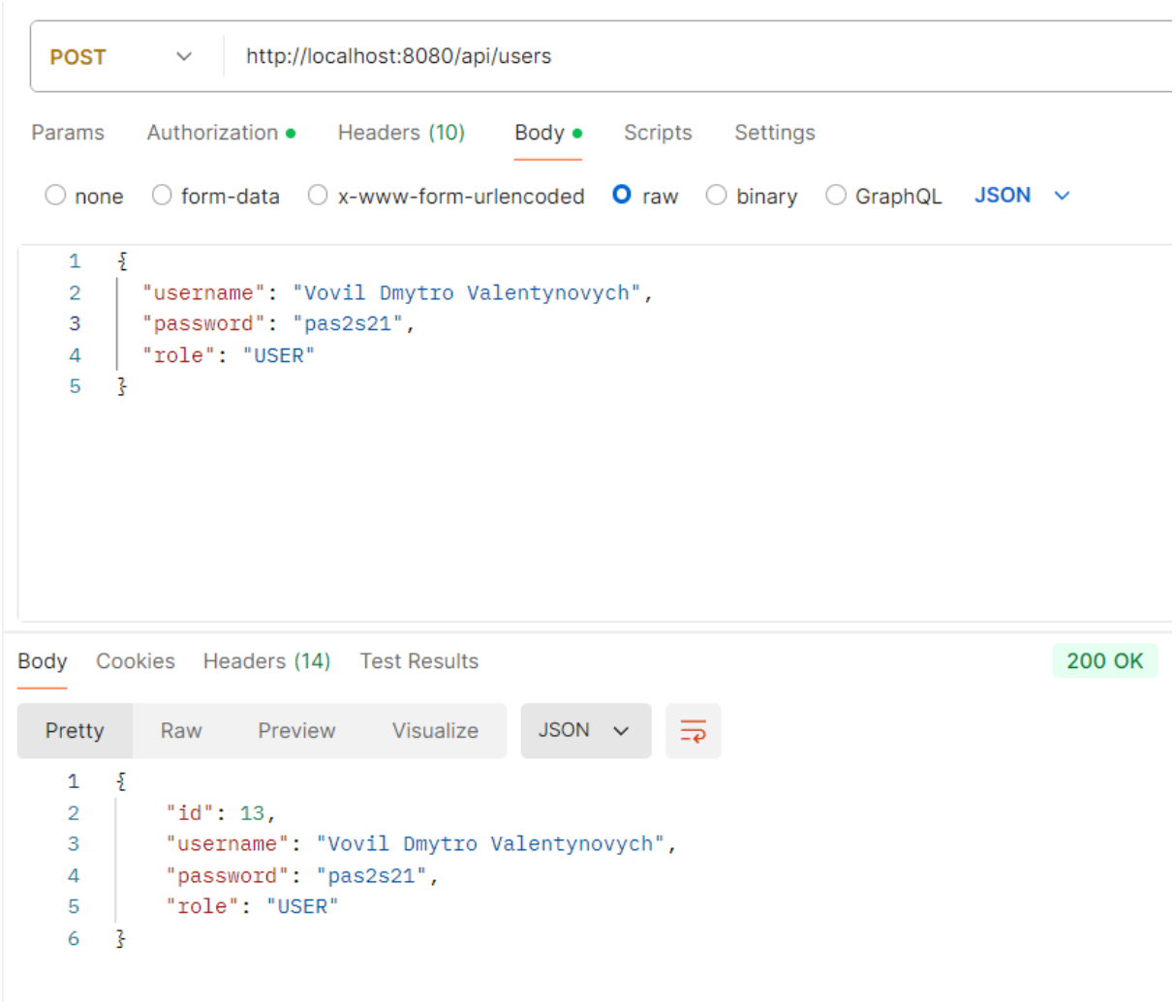

Рис.4 – Приклад створення даних користувача

Для виконання запиту для оновлення даних (PUT, UPDATE) одного користувача, потрібно використати такий шлях:

```
http://localhost:8080/api/users/2
```

На рисунку 5 показан результат введення цього шляху

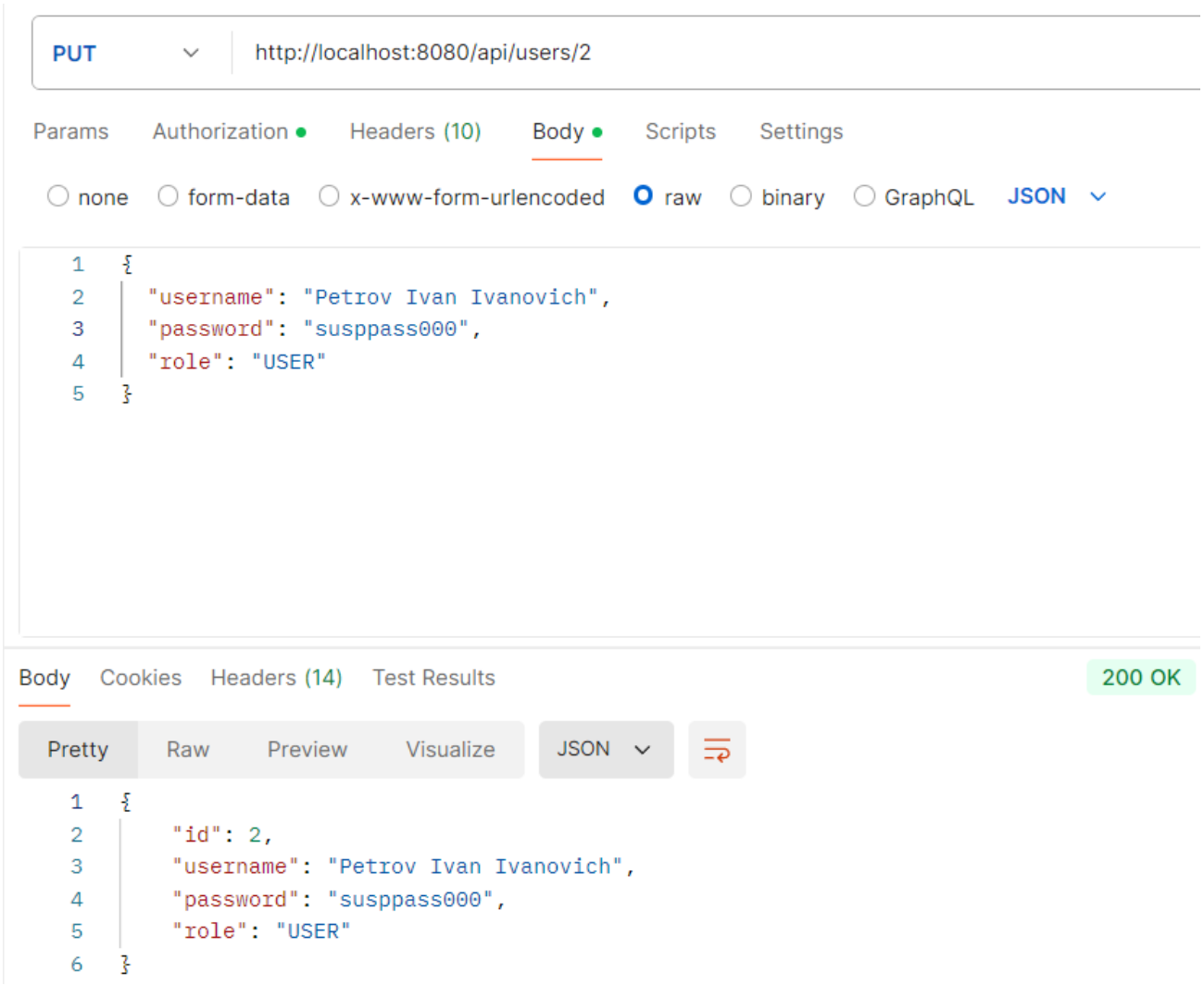

Рис.5 – Приклад оновлення даних користувача

Для виконання запиту для видалення даних (DELETE) одного користувача, потрібно використати такий шлях:

```
http://localhost:8080/api/users/13
```

На рисунку 6 показан результат введення цього шляху

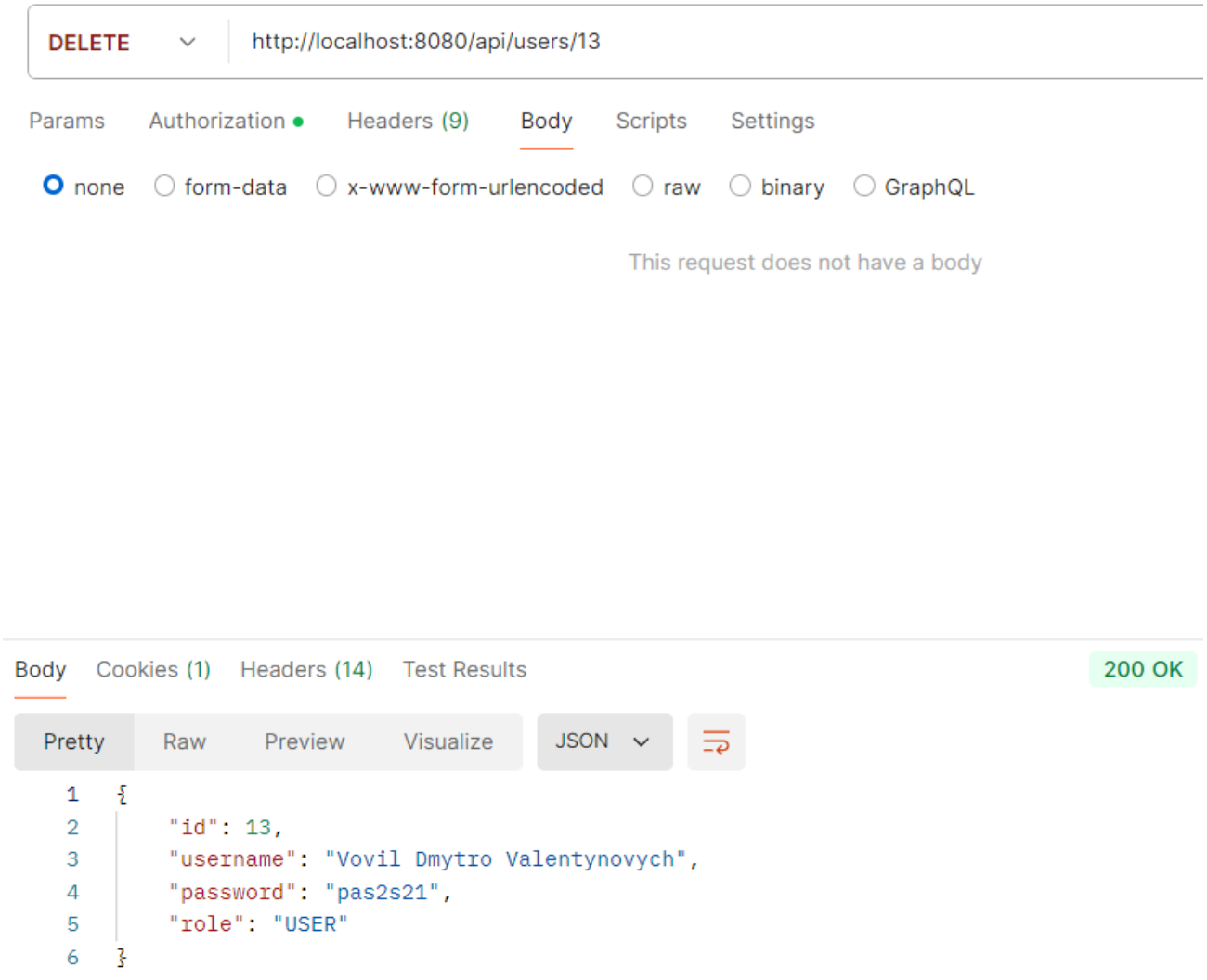

Рис.6 – Приклад видалення даних користувача за id

3 ПРОЕКТУВАННЯ І РОЗРОБКА ШАРУ СХОВИЩА

3.1. Вибір бази даних

Для цього проекту була обрана база даних **PostgreSQL**. Вибір PostgreSQL був обґрунтований кількома ключовими факторами:

- **Надійність і стійкість до великих обсягів даних**: PostgreSQL добре зарекомендувала себе як одна з найнадійніших систем керування базами даних (СКБД). Вона здатна обробляти великі обсяги даних, що особливо важливо для банківських систем, де обробка транзакцій і збереження історії операцій є критичними.
- **Підтримка транзакцій і ACID-поведінки**: PostgreSQL забезпечує надійне виконання транзакцій, з дотриманням принципів ACID (Atomicity, Consistency, Isolation, Durability), що є надзвичайно важливим для зберігання фінансових даних.
- **Гнучкість і масштабованість**: База даних легко масштабується, що дозволяє розширювати систему у випадку збільшення кількості користувачів та операцій без втрати продуктивності.
- **Сумісність із Spring Boot**: PostgreSQL має чудову інтеграцію з Spring Boot завдяки підтримці JDBC, Hibernate і JPA, що полегшує розробку і роботу з базою даних.

PostgreSQL також підтримує складні запити, роботу з JSON-форматом і має багатий набір інструментів для оптимізації роботи з базою даних, що робить її оптимальним вибором для складних корпоративних систем, таких як банківська система.

3.2. Налаштування підключення

Налаштування підключення до бази даних PostgreSQL в Spring Boot було реалізовано через конфігураційний файл **application.yml**. У цьому файлі зберігаються всі необхідні параметри для підключення до бази даних, а також налаштування для роботи з JPA і Hibernate.

Основні налаштування у файлі application.yml:

```
spring:
  application:
    name: curvova_bank
  jpa:
```

```
    database: POSTGRESQL
    show-sql: true
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        dialect: org.hibernate.dialect.PostgreSQLDialect
  datasource:
    url: jdbc:postgresql://localhost:5432/banking_system
    username: postgres
    password: 1397
    driverClassName: org.postgresql.Driver
```

Опис налаштувань:

1. **spring.jpa.database: POSTGRESQL** — вказує, що для цього додатку використовується база даних PostgreSQL.
2. **spring.jpa.show-sql: true** — параметр, що дозволяє виводити SQL-запити у консоль для полегшення налагодження.
3. **spring.jpa.hibernate.ddl-auto: update** — автоматично оновлює схему бази даних при змінах у моделях (класах-сутностях). Це дозволяє швидко тестувати нові моделі без необхідності вручну змінювати структуру таблиць.
4. **spring.datasource.url** — URL підключення до бази даних PostgreSQL. У цьому випадку використовується локальна база даних, що працює на порту 5432, і називається вона banking_system.
5. **spring.datasource.username** та **spring.datasource.password** — логін і пароль для підключення до бази даних.
6. **spring.datasource.driverClassName: org.postgresql.Driver** — вказує драйвер, який необхідний для підключення до PostgreSQL через JDBC.

На рисунку 3 показано підключення IntelliJ Idea до PostgreSQL, в самій програмі не потрібно налаштовувати підключення, воно з’єднується автоматично.

			PID	User	Application	Client	Backend start	Transaction start	State	Wait event
⊗	■	▸	4056	postgr...	PostgreSQL JDBC D...	127.0.0.1	2024-09-27 19:29:55...		idle	Client: ClientRead
⊗	■	▸	11676	postgr...	IntelliJ IDEA 2024.1.2	127.0.0.1	2024-09-27 17:48:43...		idle	Client: ClientRead

Рис. 7 – Результат підключення додатку до БД

3.3. Репозиторії

Репозиторії є важливим компонентом у Spring Data JPA, який використовується для взаємодії з базою даних на рівні об'єктів. Репозиторії дозволяють уникати написання складних SQL-запитів, забезпечуючи більш декларативний підхід до доступу до даних.

Для кожної сутності у додатку було створено окремий інтерфейс репозиторію, який успадковується від **JpaRepository**. Це надає всі стандартні CRUD-операції (створення, читання, оновлення, видалення) без необхідності додаткової реалізації. Ось приклад для репозиторію **UserRepository**, який керує операціями над сутністю користувача:

Приклад **UserRepository**:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    User findByUsername(String username);
    List<User> findByRole(String role);
}
```

У цьому репозиторії визначені стандартні методи CRUD, що автоматично надаються інтерфейсом **JpaRepository**, а також два кастомні методи:

- **findByUsername(String username)** — для пошуку користувача за ім'ям.
- **findByRole(String role)** — для пошуку користувачів за їхньою роллю.

Spring Data JPA автоматично генерує SQL-запити для цих методів на основі їхніх імен.

3.4. Моделі даних (Entity класи)

Моделі даних або сутності є класами, які представляють таблиці бази даних у додатку. У проекті було створено кілька сутностей для різних аспектів системи, таких як користувачі, транзакції, рахунки та картки.

Приклад моделі для користувача (**User.java**):

```
@Entity
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String username;
    private String password;
    private String role;

 }
```

У цьому класі:

- **@Entity** — позначає клас як сутність, що відповідає таблиці у базі даних.
- **@Table(name = "users")** — визначає, що ця сутність зберігається у таблиці users.
- **@Id** та **@GeneratedValue(strategy = GenerationType.IDENTITY)** — вказують, що поле id є первинним ключем і буде автоматично генеруватися базою даних.

Налаштування зв'язків між сутностями

У системах, таких як банківські, часто існує потреба в налаштуванні зв'язків між різними таблицями (сутностями). Для цього в Spring JPA використовуються анотації **@OneToMany**, **@ManyToOne**, **@ManyToMany** тощо, щоб відобразити зв'язки між сутностями.

@OneToMany (один до багатьох) - один екземпляр першого елемента може бути пов'язаний з безліччю екземплярів другого елементу

@ManyToOne (багато до одного) - елемент з другої сутності може бути пов'язаний лише з одним елементом першої.

@ManyToMany (багато до багатьох) - один примірник одного з елементів може бути пов'язаний з безліччю екземплярів іншого елемента і той же визначається в зворотному напрямку.

Приклад налаштування зв'язку між рахунком та користувачем:

```
@Entity
public class Account {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;

}
```

У цьому прикладі використовується зв'язок **@ManyToOne** для відображення того, що кожен рахунок належить одному користувачу. Нижче показана таблиця зв'язків між сутностями

Таблиця 3 – Список зв'язків між сутностями у банківській системі

Зв'язки між сутностями	Вид зв'язку між сутностями
Card - Account	ManyToOne
Transaction - Account	ManyToOne
Financial Product - Account	ManyToOne

4 ПРОЕКТУВАННЯ І РОЗРОБКА ЖУРНАЛЮВАННЯ

Журналювання (логування) є важливою частиною будь-якого додатку, оскільки воно дозволяє відслідковувати події, помилки та забезпечувати моніторинг роботи системи. У цьому проекті реалізація журналювання допомагає відслідковувати роботу різних компонентів додатку, таких як контролери, сервіси та репозиторії, а також допомагає виявляти та усувати помилки.

4.1. Вибір бібліотеки

Для реалізації журналювання у цьому проекті було обрано бібліотеку **Logback**, яка є стандартним рішенням для журналювання в Spring Boot. Logback має низку переваг, які зробили її оптимальним вибором для цього проекту:

- **Вбудована підтримка в Spring Boot**. Logback постачається разом зі Spring Boot як бібліотека за замовчуванням, тому не потрібно виконувати додаткових налаштувань для інтеграції.
- **Висока продуктивність**. Logback забезпечує мінімальний вплив на продуктивність додатку навіть при інтенсивному використанні журналів.

- **Гнучка конфігурація**. Вона дозволяє налаштовувати рівні логування, формати логів, способи їх збереження (у файл, у базу даних тощо), а також підтримує ротацію логів (щоб файли не росли безконтрольно).
- **Підтримка різних форматів виводу**. Логи можуть бути виведені у консоль, файл або інші цільові сховища, що спрощує моніторинг системи.

4.2. Налаштування логування

Налаштування журналювання у Spring Boot виконується через конфігураційний файл **application.yml** або **application.properties**. У нашому проекті використовувався файл **application.yml** для налаштування основних параметрів логування.

Приклад конфігурації логування у файлі application.yml:

```
logging:
  level:
    root: INFO
    com.application.cursova: DEBUG
  pattern:
    file: "%d %-5level [%thread] %logger : %msg%n"
  file:
    name: log-file.log
```

Опис конфігурації:

- **logging.level.root: INFO** — це налаштування визначає глобальний рівень логування для всього додатку. У цьому випадку встановлений рівень **INFO**, що означає, що додаток буде логувати всі події на рівні інформації, а також більш важливі події (WARN, ERROR).
- **logging.level.com.application.cursova: DEBUG** — для певного пакету додатку (com.application.cursova) встановлений рівень **DEBUG**, що дозволяє відслідковувати детальніші події, які будуть корисні для відлагодження.

- **logging.pattern.file** — визначає формат запису логів у файл. Формат містить:
  - %d — дату і час події.
  - %level — рівень події (DEBUG, INFO, WARN, ERROR).
  - %thread — потік, який викликав подію.
  - %logger — ім'я класу або пакету, який виконав логування.
  - %msg — саме повідомлення, що логувалося.
- **logging.file.name: log-file.log** — визначає ім'я файлу, у який записуються логи.

Ці налаштування дозволяють гнучко керувати логуванням у різних частинах додатку та зберігати логи у файл із налаштованим форматом.

4.3. Створення логів

Для того, щоб створювати логи у різних частинах додатку, використовується клас **Logger** з бібліотеки **SLF4J**, яка надає інтерфейс для Logback. Логи створюються у контролерах, сервісах та репозиторіях для відстеження важливих подій, помилок та інформації про процес виконання.

Приклад логування у контролері:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@RestController
@RequestMapping("/api/users")
public class UserController {

    private static final Logger logger =
LoggerFactory.getLogger(UserController.class);

    @GetMapping
    public ResponseEntity<List<User>> getAllUsers() {
```

```
        logger.info("Отримано запит на отримання всіх
користувачів");
        List<User> users = userService.getAllUsers();
        return ResponseEntity.ok(users);
    }

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable Long id)
{
        logger.debug("Отримано запит на користувача з ID: {}", id);
        User user = userService.getUserById(id);
        if (user == null) {
            logger.error("Користувач з ID {} не знайдений", id);
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(user);
    }
}
```

Опис логування у контролері:

- **logger.info()** — використовується для логування інформативних подій, наприклад, коли отримано запит на отримання всіх користувачів.
- **logger.debug()** — використовується для детальнішого відстеження подій, наприклад, для отримання ID користувача. Це корисно під час відлагодження.
- **logger.error()** — використовується для логування помилок, коли, наприклад, користувач не знайдений у базі даних.

Подібний підхід використовувався у всіх шарах додатку:

- У **сервісному шарі** для логування результатів виконання бізнес-логіки, наприклад, під час створення нового користувача або обробки транзакцій.
- У **шарі репозиторіїв** для логування взаємодії з базою даних, особливо при виконанні нестандартних запитів або під час обробки помилок бази даних.

4.4. Рівні логування

У цьому проекті використовувалися різні рівні логування для різних цілей:

- **INFO** — для логування важливих подій, які відбуваються в додатку, наприклад, успішне виконання запитів або операцій. Цей рівень є базовим і використовується для того, щоб мати загальне уявлення про те, що відбувається в додатку.
- **DEBUG** — для логування детальніших подій, які використовуються під час розробки та налагодження. Цей рівень використовується для отримання додаткової інформації про стан програми, що допомагає виявляти причини помилок або непередбачуваної поведінки.
- **ERROR** — для логування помилок, коли щось пішло не так. Це критичні події, які потребують негайної уваги, оскільки вони можуть призвести до некоректної роботи системи.

Приклад використання рівнів логування:

```
logger.info("Запит на створення нового користувача успішно
виконано");

logger.debug("Отримані дані про користувача: {}", user);

logger.error("Помилка при створенні користувача: електронна пошта
вже існує");
```

4.5. Зберігання та аналіз логів

У цьому проекті логи зберігаються у файлі, що було налаштовано у файлі application.yml. Для цього був використаний файл **log-file.log**, у якому записуються всі події, що були залоговані на рівні INFO і вище.

Переваги зберігання логів у файл:

- Логи можна зберігати протягом довгого часу і аналізувати їх у разі виникнення помилок або проблем з продуктивністю.
- Лог-файли можуть бути ротаційними, тобто їхній розмір можна обмежити, а старі файли автоматично архівувати або видаляти.

Ротація логів:

Ротація логів дозволяє контролювати розмір лог-файлів, щоб вони не зростали безмежно. Наприклад, можна налаштувати ротацію таким чином, щоб створювати новий файл кожного дня або після досягнення певного розміру.

Аналіз логів:

Для аналізу логів можна використовувати інструменти, такі як **ELK Stack** (Elasticsearch, Logstash, Kibana), які дозволяють збирати, зберігати та аналізувати логи в реальному часі. ELK Stack дозволяє здійснювати пошук і фільтрацію логів, створювати візуалізації та налаштовувати алерти на основі критичних подій у логах.

У нашому проекті лог-файли можна аналізувати вручну або використовувати інструменти для автоматизації процесу аналізу у випадку збільшення кількості подій або помилок.

Нижче продемонстровано виведені повідомлення після виконання видалення даних:

```
08:56.340+03:00  INFO c.a.cursova.aspect.LoggingAspect: Метод
сервісу викликаний: deleteUser
```

```
08:56.340+03:00  INFO c.a.cursova.aspect.LoggingAspect: Результат у
форматі JSON: {"id":13,"username":"Vovil Dmytro
Valentynovych","password":"pas2s21","role":"USER"}

08:56.341+03:00  INFO c.a.cursova.aspect.LoggingFilter:

Відповідь: статус 200
```

5 ПРОЕКТУВАННЯ І РОЗРОБКА БЕЗПЕКИ

5.1 Вибір механізму аутентифікації та авторизації

У проекті для забезпечення безпеки та контролю доступу було обрано механізм аутентифікації та авторизації на основі **Basic Auth**. Basic Authentication є одним з найпростіших механізмів для захисту API, коли клієнт передає ім'я користувача та пароль у заголовку HTTP-запиту. Цей механізм підходить для невеликих додатків або етапів розробки, де немає потреби в складних схемах аутентифікації.

Основні причини вибору **Basic Auth**:

1. **Простота впровадження**: Basic Auth не вимагає додаткових налаштувань або серверів для керування сесіями.
2. **Інтеграція з Spring Security**: Spring Security має вбудовану підтримку Basic Auth, що дозволяє легко налаштувати його використання.
3. **Підходить для API**: Basic Auth часто використовується для захисту REST API, оскільки клієнт може надсилати креденшіали в кожному запиті, що дозволяє уникати використання сесій.

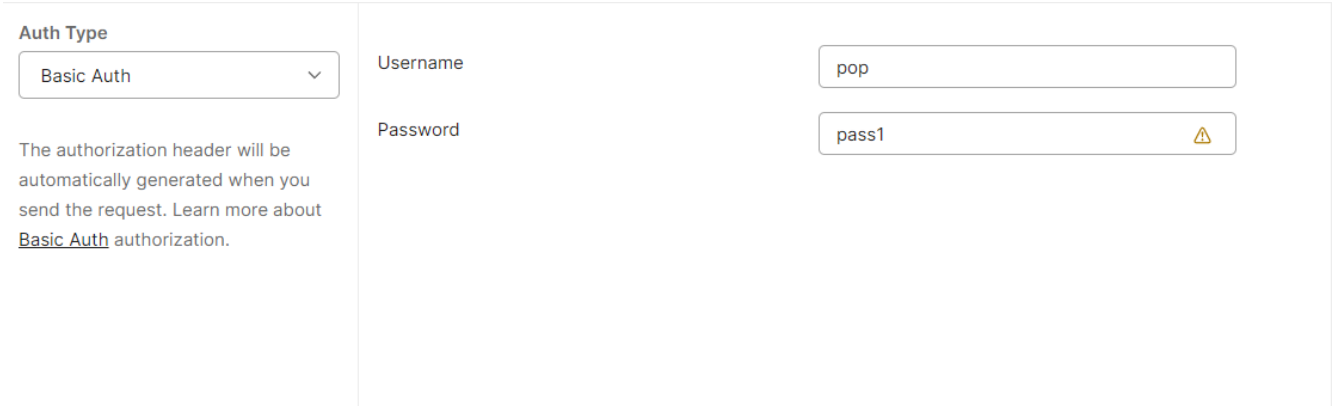

Рис. 8 – Меню при виборі безпеки Basic Auth для додатку

5.2 Налаштування Spring Security

Для налаштування безпеки у Spring Boot використовувалася бібліотека **Spring Security**, яка інтегрована в проект через конфігураційний клас **SecurityConfig.java**. У цьому класі налаштовується система безпеки, яка визначає доступ до різних ресурсів на основі ролей та методів HTTP.

Основні налаштування у класі SecurityConfig.java:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private final UserDetailsService userDetailsService;

    public SecurityConfig(UserDetailsService userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
                .csrf(csrf -> csrf.disable()) // Вимкнено CSRF-захист для API
                .authorizeHttpRequests(authorize -> authorize
                        .requestMatchers(HttpMethod.GET, "/api/**").permitAll()
// Доступ до GET-запитів без авторизації
                        .requestMatchers(HttpMethod.POST,
"/api/**").authenticated() // Авторизація для POST-запитів
                        .requestMatchers(HttpMethod.PUT,
"/api/**").authenticated() // Авторизація для PUT-запитів
```

```
                        .requestMatchers(HttpMethod.DELETE,
"/api/**").authenticated() // Авторизація для DELETE-запитів
                        .anyRequest().permitAll()
                )
                .httpBasic(withDefaults()); // Використання Basic Authentication

        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return NoOpPasswordEncoder.getInstance(); // використовується
NoOpPasswordEncoder (незашифрований пароль)
    }
}
```

Опис налаштувань:

- **@EnableWebSecurity** — ця анотація дозволяє активувати механізми безпеки у Spring.
- **SecurityFilterChain** — це основний механізм, який відповідає за фільтрацію запитів та авторизацію доступу до ресурсів.
- **Налаштування доступу до API**:
  - **GET-запити** на всі шляхи, що починаються з **/api/** дозволяються без авторизації.
  - **POST, PUT, DELETE-запити** вимагають аутентифікації.
- **httpBasic()** — включає Basic Authentication для захисту API.
- **Паролі** не шифруються для спрощення тестування на етапі розробки через використання **NoOpPasswordEncoder**, але у продуктивному середовищі цей механізм має бути замінений на **BCryptPasswordEncoder** або інший захищений енкодер.

5.3 Реалізація аутентифікації

Процес аутентифікації у додатку базується на використанні **UserDetailsService**, який забезпечує отримання інформації про користувача з бази даних або іншого джерела. Цей сервіс інтегрується з Spring Security для

перевірки автентичності користувачів на основі їхніх креденшіалів (логіну та паролю).

Приклад реалізації CustomUserDetailsService:

```
@Service
public class CustomUserDetailsService implements UserDetailsService
{

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException("Користувач не
знайдений");
        }
        return new
org.springframework.security.core.userdetails.User(user.getUsername
(), user.getPassword(), new ArrayList<>());
    }
}
```

Опис:

- **CustomUserDetailsService** — це клас, який реалізує інтерфейс **UserDetailsService**, необхідний для інтеграції з Spring Security. Він завантажує користувача з бази даних за його іменем користувача.
- Якщо користувача не знайдено, кидається виключення **UsernameNotFoundException**.
- У цьому прикладі роль користувача поки що не налаштовується (передається порожній список), але це можна легко розширити для реалізації авторизації на основі ролей.

Приклад реалізації введення даних з аутентифікацією прав користувача

Наприклад, якщо нам потрібно змінити дані для сутності Users, будемо використовувати такі дані:

```
{
  "username": "Akhtemenko Denys Viktorovych",
  "password": "denspass2709",
  "role": "USER"
}
```

Далі, якщо в якості типу аутентифікації ми вибрали вкладку **No Auth,** то ми отримаємо такий результат на рисунку 9. Результат виявився помилковим, тому що в файлі **SecurityConfig** ми налаштували дані так, що запити у яких потрібно змінювати, видаляти, додавати дані користувача повинен тільки користувач з роллю **ADMIN.**

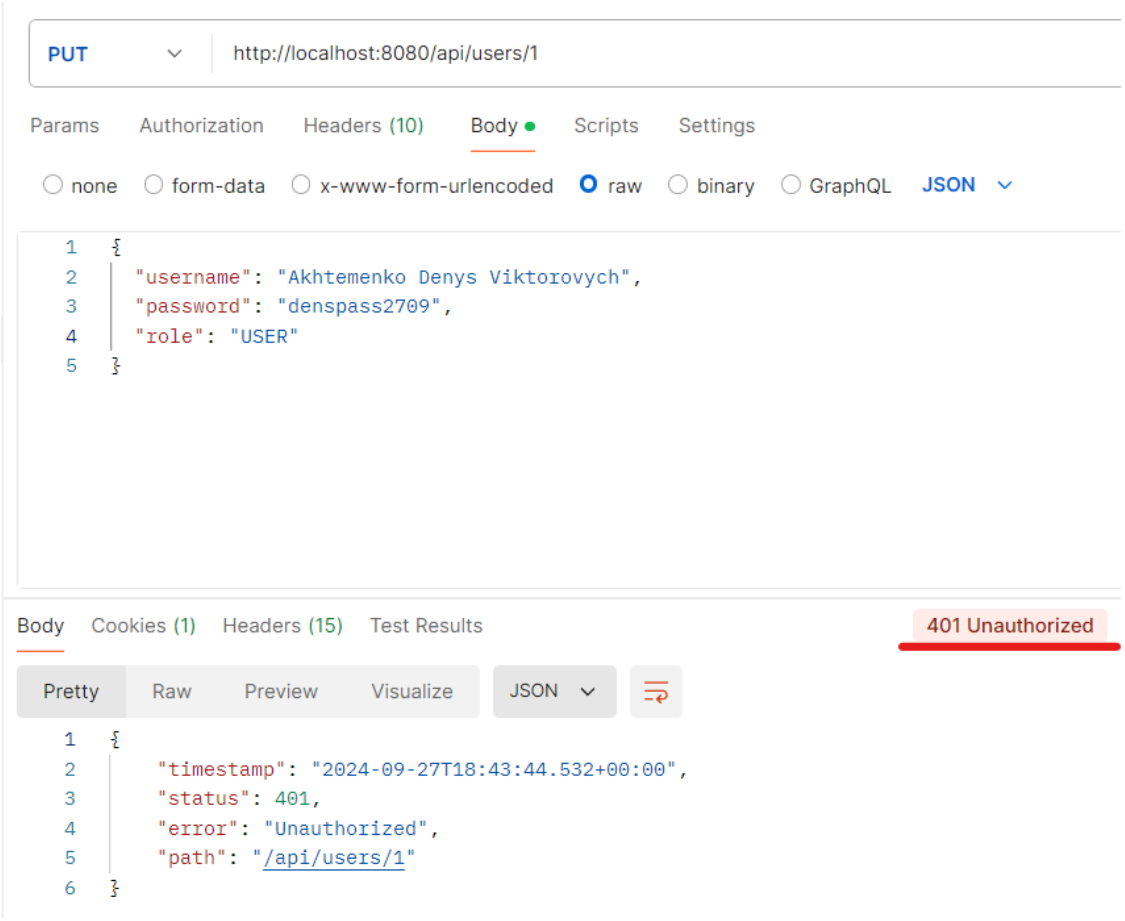

Рис. 9 – Результат виконання запиту без аутентифікації

Тепер повторимо обробку запиту, але в вкладці тип аутентифікації виберемо вкладку **Basic Auth.** Для корректного виконання запиту будемо використовувати користувача з ім'ям “**pop**”, так як він має роль **ADMIN.** Результат виконання запиту продемонстровано на рисунку 10.

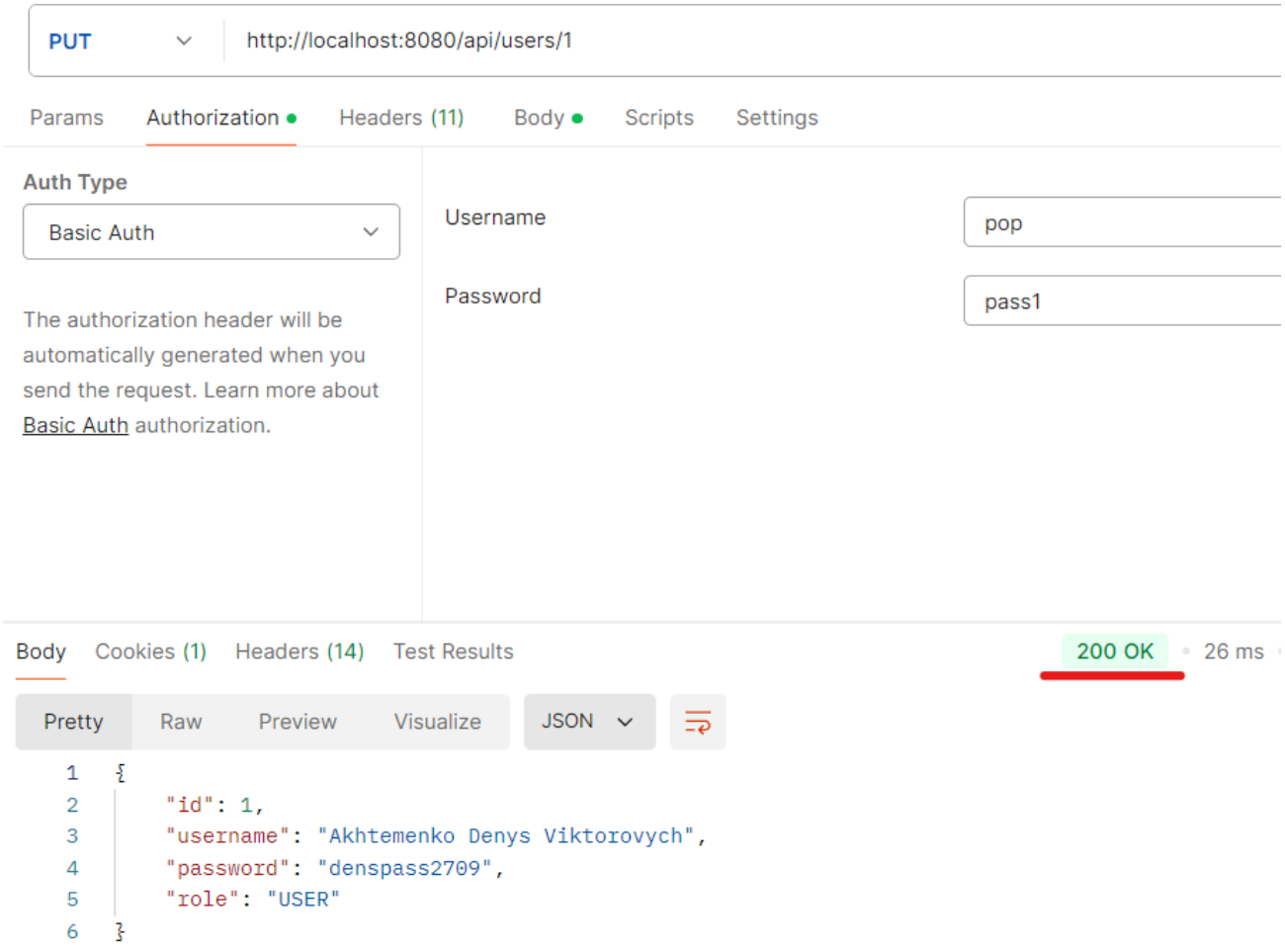

Рис. 10 – Результат виконання запиту з аутентифікацією

5.4 Реалізація авторизації

Авторизація у додатку реалізована на основі ролей користувачів. Авторизація дозволяє визначити, які ресурси додатку можуть бути доступні для різних користувачів, залежно від їхніх ролей та прав доступу. У Spring Security авторизація налаштовується через механізм контролю доступу до HTTP-запитів у класі **SecurityConfig**.

Реалізація авторизації на основі HTTP-запитів:

- **requestMatchers(HttpMethod.GET, "/api/**").permitAll()** — дозволяє доступ до всіх GET-запитів без авторизації.
- **requestMatchers(HttpMethod.POST, "/api/**").authenticated()** — вимагає авторизації для POST-запитів.
- **requestMatchers(HttpMethod.PUT, "/api/**").authenticated()** — вимагає авторизації для PUT-запитів.
- **requestMatchers(HttpMethod.DELETE, "/api/**").authenticated()** — вимагає авторизації для DELETE-запитів.

Авторизація на основі ролей може бути розширена за допомогою додавання ролей для користувачів та використання анотацій **@PreAuthorize** або конфігураційного методу **hasRole()**, що дозволяє обмежувати доступ до конкретних ресурсів на основі ролей.

ВИСНОВКИ

У цій курсовій роботі було розроблено банківську систему на основі Java Spring, де основну увагу приділено модульності, гнучкості та надійності.

Для створення веб-шару використано Spring Boot, що дозволило швидко налаштувати REST API з мінімальними зусиллями. Сервісний шар було розроблено як окремі класи, відповідальні за різні аспекти системи, такі як управління користувачами, рахунками та транзакціями. Це допомогло забезпечити чітке розділення бізнес-логіки та полегшити підтримку системи.

Для роботи з даними використано PostgreSQL та Spring Data JPA, що забезпечило надійність і легкість взаємодії з базою даних через репозиторії, без необхідності писати SQL-запити вручну.

Журналювання реалізовано з використанням Logback, що дозволило відстежувати важливі події та помилки в додатку, забезпечуючи надійний механізм моніторингу. Для захисту системи було налаштовано Spring Security з використанням Basic Auth, що гарантувало аутентифікацію користувачів та захищений доступ до ресурсів.

Розроблена система є стійкою, безпечною та легкою у розширенні, що відповідає вимогам до сучасних банківських додатків.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

“Що таке Java і де вона використовується” - https://goit.global/ua/articles/shcho-take-java-i-de-vona-vykorystovuietsia/

«Освоюємо Java» - https://uk.wikibooks.org/wiki/%D0%9E%D1%81%D0%B2%D0%BE%D1%8E%D1%94%D0%BC%D0%BE_Java

“Фреймворк Spring та його особливості” - https://highload.today/uk/frejmvork-spring-ta-jogo-osoblivosti/

«Spring для лінивих. Основи, базові концепції» - https://javarush.com/ua/groups/posts/uk.476.spring-dlja-lnivikh-osnovi-bazov-koncepc-ta-prikladi-z-kodom-chastina-1

«Фреймворк Spring» - https://proit.ua/frieimvork-spring-prostii-manual-vid-rozrobnika-z-nix/

«Що таке PostgreSQL і де воно використовується» - https://foxminded.ua/postgresql-shcho-tse/

«Переваги та недоліки використання Spring Boot» - https://javarush.com/ua/groups/posts/uk.3380.kava-breyk-75-perevagi-ta-nedolki-vikoristannja-spring-boot-funkc-dlja-rjadkv-u-java