

Міністерство освіти і науки України
Національний університет «Одеська політехніка»
Навчально-науковий інститут штучного інтелекту та робототехніки
Кафедра штучного інтелекту та аналізу даних

Маляренко Анастасія Олександрівна,
студентка групи AI-233

КУРСОВА РОБОТА

«Система управління автосервісом.»
з дисципліни «Об'єктно-орієнтоване програмування»

Спеціальність:
122 Комп'ютерні науки

Освітня програма: Комп'ютерні науки

Керівник:
Годовиченко Микола Анатолійович,
кандидат технічних наук, старший викладач

Одеса – 2025

Національний університет «Одеська політехніка»
Навчально-науковий інститут комп'ютерних систем

АНОТАЦІЯ

Маляренко А.О. Розробка серверного застосунку для обліку обслуговування автомобілів : курсова робота з дисципліни «Об'єктно-орієнтоване програмування» за спеціальністю «122 Комп'ютерні науки» / Маляренко Анастасія Олександрівна; керівник Годовиченко Микола Анатолійович. – Одеса : Нац. ун-т «Одес. політехніка», 2025. – 58 с.

Курсова робота містить основну текстову частину на 49 сторінках, список використаних джерел із 5 найменувань на 1 сторінці, а також 25 сторінок додатків з фрагментами коду.

У курсовій роботі реалізовано серверний вебзастосунок для обліку клієнтів, їхніх автомобілів, механіків, видів послуг та записів технічного обслуговування. Для реалізації проєкту використано мову програмування Java, фреймворк Spring Boot, засоби Spring Security, JWT та OAuth2 для автентифікації. Сховище даних реалізовано за допомогою Spring Data JPA та реляційної бази даних. Реалізовано повноцінний REST API з підтримкою CRUD-операцій, а також систему ролей для клієнтів та адміністраторів. Система протестована з використанням Postman.

Ключові слова: Java, Spring Boot, REST API, обслуговування автомобілів, Spring Security, база даних.

ЗМІСТ

Вступ.....	4
1 Аналіз предметної області та постановка задачі.....	6
1.1 Постановка задачі.....	6
1.2 Опис предметної області.....	6
1.3 Сценарії використання.....	9
2 Проєктування програмного забезпечення.....	10
2.1 Структура даних: сутності та зв'язки.....	10
2.2 Архітектура застосунку (Controller–Service–Repository).....	11
2.3 REST API: опис реалізованих запитів.....	12
3 Реалізація програмного продукту.....	14
3.1 Моделі (entity-класи).....	14
3.2 Репозиторії.....	17
3.3 Сервіси.....	18
3.4 Контролери.....	20
3.5 Конфігурація безпеки (Spring Security, JWT, OAuth2).....	23
4 Тестування REST API.....	27
4.1 Тестування автентифікації користувача.....	27
4.2 Тестування клієнтських операцій.....	29
4.3 Тестування автомобілів клієнтів.....	32
4.4 Тестування операцій з механіками.....	35
4.5 Тестування записів на обслуговування.....	36
4.6 Тестування типів послуг.....	40
4.7 Тестування статистики та звітності.....	41
Загальні висновки.....	45
Список використаних джерел.....	48
Додатки	49

ВСТУП

У сучасному цифровому суспільстві інформаційні технології відіграють важливу роль у забезпеченні ефективної роботи підприємств, організацій та сервісних служб. Зокрема, галузь технічного обслуговування транспортних засобів вимагає сучасних засобів автоматизації для обліку клієнтів, їхніх автомобілів, запланованих і виконаних робіт, а також контролю за працею персоналу. В умовах постійного зростання кількості автомобілів та сервісних центрів виникає необхідність у створенні програмних рішень, які дозволяють централізовано, швидко та безпомилково обробляти великі обсяги даних, пов'язаних із обслуговуванням автотранспорту.

Об'єктно-орієнтоване програмування (ООП) як сучасна парадигма розробки програмного забезпечення забезпечує надійний фундамент для створення масштабованих, модульних та розширюваних інформаційних систем. Завдяки використанню принципів інкапсуляції, наслідування, поліморфізму та абстракції, об'єктно-орієнтовані системи дозволяють ефективно моделювати складні предметні області, а також забезпечувати повторне використання коду та підтримку великих проєктів.

Метою даної курсової роботи є розробка серверного застосунку для обліку клієнтів, автомобілів, механіків, видів послуг і сервісних записів із використанням мови програмування Java та фреймворку Spring Boot. Система повинна реалізовувати автентифікацію користувачів (через логін-пароль, JWT та OAuth2), підтримку ролей (користувач, адміністратор), REST API з повним переліком CRUD-операцій для кожної сутності, а також збереження даних у реляційній базі за допомогою JPA та Hibernate.

У рамках проєкту було реалізовано логічну архітектуру застосунку, яка включає окремі шари моделі, сервісу, контролерів та репозиторіїв. Усі компоненти взаємодіють відповідно до принципів інверсії управління та впровадження залежностей, що значно підвищує гнучкість і тестованість застосунку. Для забезпечення безпеки даних реалізовано механізми автентифікації та авторизації за

допомогою Spring Security, включно з генерацією JWT-токенів та інтеграцією з Google OAuth2.

Також у ході роботи було протестовано працездатність основного API за допомогою Postman та Swagger UI, що дозволило підтвердити правильність реалізації запитів і відповідей, перевірити валідацію введених даних, а також виявити і усунути помилки. Застосунок відповідає стандартам REST-архітектури, забезпечує належний рівень зручності використання та може бути легко розгорнутий на віддаленому сервері (Render, Heroku тощо).

Таким чином, курсова робота охоплює весь цикл створення серверного застосунку – від проєктування об'єктної моделі до тестування готового API. Вона не лише демонструє практичне застосування знань з ООП, але й формує навички, необхідні для професійної розробки сучасних backend-рішень. Отриманий досвід стане корисною основою для майбутньої роботи в IT-сфері та вивчення більш складних технологій розробки програмного забезпечення.

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Постановка задачі

Необхідно створити веб-застосунок із REST-інтерфейсом, що забезпечить:

1) аутентифікацію та авторизацію користувачів із різними ролями (адміністратор, сервісний інженер, клієнт) за допомогою логін/пароль, JWT-токенів та OAuth2;

2) CRUD-операції для основних сутностей:

- Client (Клієнт);
- Car (Автомобіль);
- Mechanic (Механік);
- ServiceType (Тип послуги);
- ServiceRecord (Сервісний запис).

3) розмежування прав доступу: користувачі з роллю ADMIN можуть керувати довідниковими даними й користувачами, ролі USER — створювати та переглядати записи про обслуговування;

4) збереження даних у реляційній базі через Spring Data JPA та Hibernate;

5) документування API за допомогою Swagger UI та тестування запитів через Postman.

Результатом має стати надійний, масштабований та захищений сервіс, який зменшує час обслуговування і підвищує прозорість бізнес-процесів у сервісному центрі.

1.2 Опис предметної області «Облік технічного обслуговування автомобілів»

Розгорнута інформація про сферу обслуговування автомобілів

Сфера технічного обслуговування (ТО) транспортних засобів охоплює широкий спектр послуг — від базової діагностики до складного ремонту двигунів, електроніки, ходової частини. В умовах постійного зростання кількості автомобілів зростає й навантаження на автосервісні підприємства. Такі організації

зіштовхуються з проблемами ведення обліку клієнтів, графіків робіт, спеціалізацій працівників, повторних звернень, а також історії обслуговування кожного окремого автомобіля.

Історія та тенденції розвитку

Раніше облік у подібних закладах здійснювався вручну, на папері або в Excel-таблицях. Це призводило до значних втрат часу, помилок і дублювання даних. У сучасних умовах більшість сервісів переходять до впровадження цифрових рішень, що дозволяють автоматизувати облік і спростити роботу персоналу.

Як зазначено в дослідженні Fixico, впровадження AI-інструментів і цифрових платформ дозволяє сервісам адаптуватися до розумних авто й електромобілів, а також забезпечити ефективну взаємодію між клієнтом, сервісом і страховими компаніями [1].

Технічне обслуговування є циклічним процесом, і правильне ведення історії сервісних записів — критично важливе для безпеки транспортного засобу та зручності клієнта. Саме тому впровадження інформаційної системи з центральною базою даних є актуальним завданням для галузі.

Предметна область включає:

- клієнтів і їхні контактні дані;
- автомобілі, прив'язані до клієнтів;
- механіків із конкретною спеціалізацією;
- типи сервісних робіт (діагностика, ремонт, заміна деталей тощо);
- облік кожного факту обслуговування (дата, опис, відповідальні особи, тип послуги, транспортний засіб).

1) проблеми предметної області:

- відсутність централізованої бази даних для швидкого доступу до історії обслуговування;
- неможливість відстежити продуктивність механіків та обсяг виконаних робіт;
- дублювання записів через ручне введення;
- витрата часу на пошук інформації;
- низький рівень безпеки даних клієнтів і автомобілів;

- відсутність контролю над призначеними сервісами (наприклад, накладення записів на одну дату) [2].

2) навіщо потрібна база даних:

- для зберігання структурованої, перевіреної інформації про всі процеси в сервісі;
- для спрощення доступу до історії обслуговування автомобіля;
- для автоматизації CRUD-операцій і валідації даних;
- для реалізації захищеного доступу до інформації через систему ролей;
- для можливості аналітики — підрахунку кількості звернень, завантаженості персоналу, популярних видів сервісів.

3) яку інформацію повинна зберігати система:

- клієнти (Client): ПІБ, email, телефон, логін, пароль, роль;
- автомобілі (Car): марка, модель, рік випуску, VIN, власник;
- механіки (Mechanic): ПІБ, спеціалізація;
- типи послуг (ServiceType): назва, опис, орієнтовна вартість;
- сервісні записи (ServiceRecord): автомобіль, механік, тип послуги, дата, опис виконаних робіт.

4) як система полегшить роботу кінцевого користувача:

- автоматизація введення даних: замість ручного обліку використовується структура з перевітками;
- захист доступу: система аутентифікації та авторизації гарантує, що тільки уповноважені користувачі мають доступ до певних дій;
- пошук та фільтрація: REST API дозволяє фільтрувати записи за автомобілем, клієнтом, механіком, датою, типом послуги;
- контроль конфліктів: система дозволяє уникати дублювання записів або призначення двох сервісів на один час;
- покращена аналітика: адміністратори можуть переглядати статистику звернень, популярних послуг, навантаження на персонал.

1.3 Сценарії використання

Для повноцінного розуміння логіки функціонування інформаційної системи необхідно проаналізувати взаємодію між кінцевими користувачами та функціональними компонентами застосунку. У системі передбачено кілька типів користувачів (акторів), кожен з яких має власний набір дозволених дій. Це дозволяє реалізувати механізми безпеки, обмеження доступу та персоналізований інтерфейс.

Для ідентифікації вимог та функціоналу системи сформовано основні Use Case, представлені в таблиці 1.1.

Таблиця 1.1 – Основні сценарії використання системи різними акторами

Актор	Сценарій використання
Admin	– Створення, редагування, видалення облікових записів користувачів
	– Додавання та редагування типів послуг
	– Перегляд статистики обслуговування
	– Повний доступ до сервісних записів, фільтрація за автомобілем, механіком, датою
User	– Реєстрація нового облікового запису
	– Аутентифікація в системі
	– Додавання та редагування даних про автомобілі
	– Створення сервісного запису для вибраного автомобіля
	– Перегляд історії обслуговування власного транспорту
API	– Виконання REST-запитів для всіх CRUD-операцій з використанням авторизації
	– Інтеграція з фронтендом або зовнішніми сервісами через документоване API

Таке структурування взаємодії дозволяє ефективно розподілити функціональність системи та забезпечити її гнучкість, масштабованість і безпеку.

ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Структура даних: сутності та зв'язки

Розроблена система базується на об'єктно-орієнтованому підході, що передбачає чітке розділення логіки на сутності (Entity), які безпосередньо відповідають елементам предметної області. Основними сутностями системи є:

- Client — клієнт автосервісу, що зберігає персональні дані та пов'язаний з користувачем системи;
- Car — автомобіль клієнта, що містить характеристики (марку, модель, рік, VIN);
- Mechanic — співробітник сервісу з визначеною спеціалізацією;
- ServiceType — вид послуги, що має назву, опис, вартість;
- ServiceRecord — запис про обслуговування, який зв'язує автомобіль, механіка, тип послуги, дату й опис дій.

Визначені типи зв'язків між сутностями представлено в таблиці 2.1.

Таблиця 2.1 – Типи зв'язків між сутностями

Сутність	Тип зв'язку	Сутність
Client	1 : N	Car
Car	1 : N	ServiceRecord
Mechanic	1 : N	ServiceRecord
ServiceType	1 : N	ServiceRecord

Пояснення розстановки зовнішніх ключів відповідно до типів зв'язків:

1) Client – Car [1 : N]

- один клієнт може володіти кількома автомобілями;
- у таблиці Car міститься зовнішній ключ `client_id`, який посилається на `id` таблиці Client.

2) Car – ServiceRecord [1 : N]

- один автомобіль може мати кілька записів про обслуговування;

- у таблиці ServiceRecord зовнішній ключ car_id посилається на id таблиці Car.

3) Mechanic – ServiceRecord [1 : N]

- один механік може обслуговувати багато автомобілів;
- у таблиці ServiceRecord зовнішній ключ mechanic_id посилається на id таблиці Mechanic.

4) ServiceType – ServiceRecord [1 : N]

- один тип послуги може фігурувати у багатьох записах;
- у таблиці ServiceRecord зовнішній ключ service_type_id посилається на id таблиці ServiceType.

Ці зв'язки забезпечують цілісність даних і дають змогу ефективно організувати структуру зберігання та взаємодії сутностей у базі даних.

2.2 Архітектура застосунку (Controller–Service–Repository)

Проект дотримується принципу розділення відповідальності (Separation of Concerns), який реалізовано через трирівневу архітектуру:

1) контролери (Controller) — відповідають за обробку HTTP-запитів та повернення відповідей. Вони взаємодіють із сервісами та конвертують вхідні дані з JSON у DTO;

2) сервіси (Service) — містять бізнес-логіку. Наприклад, ClientService виконує перевірку унікальності username та email, кодує паролі, викликає збереження об'єктів;

3) репозиторії (Repository) — використовують Spring Data JPA для взаємодії з базою даних. Вони містять методи пошуку, збереження, оновлення та видалення об'єктів.

Кожен рівень є незалежним і легко тестується. Крім того, завдяки впровадженню залежностей через анотацію @RequiredArgsConstructor, система залишається гнучкою та масштабованою.

Безпека реалізована через Spring Security з підтримкою JWT і OAuth2. Для фільтрації запитів використовується JwtAuthFilter, який перевіряє токен перед кожною дією.

2.3 REST API: опис реалізованих запитів

Ендпоінти автентифікації:

- POST /api/auth/register — реєстрація нового користувача;
- POST /api/auth/authenticate — автентифікація користувача, повернення JWT;
- GET /api/auth/oauth2/success — обробка входу через OAuth2 (наприклад, Google).

Ендпоінти клієнтів:

- POST /api/clients — створення нового клієнта;
- GET /api/clients — отримання списку всіх клієнтів;
- PUT /api/clients/{id} — оновлення клієнта за ID;
- DELETE /api/clients/{id} — видалення клієнта.

Ендпоінти автомобілів:

- POST /api/cars — додавання автомобіля;
- GET /api/cars — список усіх авто;
- GET /api/cars/{id} — отримання конкретного авто;
- PUT /api/cars/{id} — редагування авто;
- DELETE /api/cars/{id} — видалення авто.

Ендпоінти механіків:

- POST /api/mechanics — додавання механіка;
- GET /api/mechanics — перегляд усіх;
- PUT /api/mechanics/{id} — редагування;
- DELETE /api/mechanics/{id} — видалення.

Ендпоінти сервісних записів:

- POST /api/records — створення запису;
- GET /api/records — список усіх записів;

- GET /api/records/car/{carId} — записи по авто;
- GET /api/records/mechanic/{mechanicId} — записи по механіку;
- PUT /api/records/{id} — редагування запису;
- DELETE /api/records/{id} — видалення.

Ендпоінти типів послуг:

- POST /api/service-types — створення нового типу послуги;
- GET /api/service-types — перегляд усіх типів послуг.

Статистика та звітність:

- GET /api/statistics/popular-services — найчастіші послуги;
- GET /api/statistics/most-serviced-cars — автомобілі з найбільшою кількістю сервісів;
- GET /api/statistics/masters/{id} — статистика майстра.

Усі захищені запити вимагають JWT-токен у заголовку Authorization: Bearer <TOKEN>.

Всі ендпоінти будуть протестовані в наступних роділах через Postman за допомогою послідовних запитів, наведених у розділі додатків.

РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Моделі (entity-класи)

В Spring Boot моделі даних реалізуються як сутності (entity-класи), які відображаються на таблиці бази даних за допомогою JPA. Такі класи позначаються анотацією `@Entity`, що вказує на віднесення класу до сутностей бази. За замовчуванням ім'я таблиці відповідає імені класу, але його можна змінити за допомогою анотації `@Table(name = "ім'я_таблиці")`. У самому класі кожне поле відповідає стовпцю в таблиці: первинний ключ позначається анотацією `@Id`, а зростаючий ідентифікатор – ще й `@GeneratedValue` з вибором стратегії генерації [3]. Наприклад:

```
@Entity
```

```
@Table(name = "clients")
```

```
public class Client {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    @Column(nullable = false)
```

```
    private String firstName;
```

```
    @Column(nullable = false)
```

```
    private String lastName;
```

```
    @Column(unique = true, nullable = false)
```

```
    private String email;
```

```
    @OneToMany(mappedBy = "owner")
```

```
    private List<Car> cars;
```

```
// Геттери/сеттери, конструктори
}
```

У цьому прикладі клас Client є сутністю, анотація @Table вказує, що вона відображається на таблицю clients. Поле id — первинний ключ з автоматичною генерацією (стратегія IDENTITY). Поля firstName, lastName, email відображаються на відповідні стовпці (за замовчуванням імена полів іменам стовпців), а унікальність поля email забезпечує параметр unique=true.

Типові проєктні сутності можуть бути, наприклад, Car, ServiceRecord, ServiceType, Mechanic. Зв'язки між ними визначаються анотаціями JPA. Наприклад, один клієнт може мати багато авто, тому зв'язок між Client і Car — “один-до-багатьох” (@OneToMany та @ManyToOne):

```
@Entity
@Table(name = "cars")
public class Car {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String make;
    private String model;

    @ManyToOne
    @JoinColumn(name = "client_id", nullable = false)
    private Client owner;

    @OneToMany(mappedBy = "car")
    private List<ServiceRecord> serviceRecords;
```

```

    // Геттери/сеттери, конструктори
}

@Entity
@Table(name = "service_records")
public class ServiceRecord {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private LocalDate serviceDate;
    private String description;

    @ManyToOne
    @JoinColumn(name = "car_id", nullable = false)
    private Car car;

    @ManyToOne
    @JoinColumn(name = "mechanic_id")
    private Mechanic mechanic;

    @ManyToOne
    @JoinColumn(name = "service_type_id")
    private ServiceType serviceType;

    // Геттери/сеттери, конструктори
}

```

У цих прикладах ми демонструємо, як класи сутностей містять інформацію про зв'язки (наприклад, @ManyToOne в Car і ServiceRecord) та основні поля. Кожний клас є простим POJO з анотаціями JPA і, можливо, Lombok для

геттерів/сеттерів. Додавання анотацій ніяк не впливає на звичайну роботу класу, але забезпечує Spring Data JPA можливість згенерувати код репозиторію за схемою бази даних.

3.2 Репозиторії

Для роботи з базою даних використовується шар репозиторіїв Spring Data JPA. Згідно з документацією, “*Spring Data JPA надає підтримку репозиторіїв для JPA*”, що спрощує розробку, надаючи готові CRUD-методи [4]. Кожна сутність зазвичай має відповідний інтерфейс репозиторію, який розширює `JpaRepository<T, ID>`. Наприклад:

```
@Repository
```

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

```
@Repository
```

```
public interface CarRepository extends JpaRepository<Car, Long> {  
    List<Car> findByMake(String make);  
    }
```

Розширення `JpaRepository` автоматично додає стандартні методи на кшталт `save()`, `findById()`, `findAll()`, `deleteById()` тощо. Опис інтерфейсу двома фразами: “По-перше, наслідуючи `JpaRepository`, ми отримуємо набір універсальних CRUD-методів (зберегти, видалити і т. д.)”. По-друге, Spring сканує такий інтерфейс і створює бін-реалізацію для нього. Завдяки цьому в коді сервісів чи контролерів можна напряму викликати `clientRepository.save(client)` без ручного опису SQL-запитів.

Крім стандартних методів, Spring Data дозволяє оголошувати власні query-методи за іменами (наприклад, `findByEmail()`) або з використанням анотації `@Query`. У наведеному прикладі `CarRepository` додано метод `findByMake`, який

автоматично вибірково знайде авто за маркою. Таким чином, репозиторій відповідає за абстрагування рівня доступу до БД: інтерфейс (позначений `@Repository`) задає модель доступу, а Spring Data JPA генерує підставну реалізацію репозиторію за допомогою `SimpleJpaRepository`.

3.3 Сервіси

На рівні сервісів (Service layer) реалізується бізнес-логіка програми та валідація даних. Шар сервісів розташовується між контролерами і репозиторіями і інкапсулює операції з сутностями [5]. Як зазначено в патерні «Service Layer», це шар, що *“визначає межі програми і набір доступних операцій... інкапсулює бізнес-логіку, координуючи транзакції і відповіді при реалізації операцій”*. У Spring-контексті класи сервісів позначаються анотацією `@Service` і зазвичай включають в собі валідацію, перетворення DTO<->Entity та виклики репозиторіїв. Як пишуть у Baeldung, сервісний шар «сприяє комунікації між контролером і шаром збереження даних, містить бізнес-логіку і, зокрема, логику валідації».

Приклад сервісу для роботи з клієнтами (ClientService) може виглядати так:

`@Service`

```
public class ClientService {
```

```
    @Autowired
```

```
    private ClientRepository clientRepository;
```

```
    public Client createClient(Client client) {
```

```
        // Приклад валідації: перевіримо унікальність email
```

```
        if (clientRepository.existsByEmail(client.getEmail())) {
```

```
            throw new IllegalArgumentException("Email уже використовується");
```

```
        }
```

```
        return clientRepository.save(client);
```

```
    }
```

```

public Client updateClient(Long id, Client clientData) {
    Client existing = clientRepository.findById(id)
        .orElseThrow(() -> new NoSuchElementException("Клієнт не
знайдений"));
    // Оновлюємо поля
    existing.setFirstName(clientData.getFirstName());
    existing.setLastName(clientData.getLastName());
    // ... інші поля
    return clientRepository.save(existing);
}
}

```

У цьому прикладі метод `createClient` перевіряє унікальність email перед збереженням (додаткова бізнес-логіка), а потім викликає `save()` репозиторію для створення запису. Метод `updateClient` отримує існуючого клієнта, виконує необхідні оновлення полів і знову зберігає. Сервіс може бути анотований `@Transactional`, якщо потрібно забезпечити транзакційність.

Інший приклад – метод оновлення запису сервісу авто (`ServiceRecord`):

```

@Service
public class ServiceRecordService {
    @Autowired
    private ServiceRecordRepository recordRepository;

    public ServiceRecord updateServiceRecord(Long id, ServiceRecord
recordData) {
        ServiceRecord record = recordRepository.findById(id)
            .orElseThrow(() -> new NoSuchElementException("Запис не
знайдений"));
        record.setDescription(recordData.getDescription());
        record.setServiceDate(recordData.getServiceDate());
    }
}

```

```
// Можна перевірити, чи механік чи тип послуги існують тощо
return recordRepository.save(record);
}
}
```

Таким чином, шар сервісів ізолює логіку: перевірки, трансформації, обробки виключень. Контролери передають прості об'єкти (DTO або сутності) сервісам, які потім повертають результат виконання (часто також сутність або DTO). Такий підхід робить контролери “легкими” (відповідальність за бізнес-правила в сервісі) та полегшує тестування і модульне розширення системи.

3.4 Контролери

Клас контролера відповідає за обробку HTTP-запитів у Spring MVC. У Spring контролери позначаються анотацією `@Controller` (якщо повертають view) або `@RestController` (для REST-сервісів), що поєднує `@Controller` та `@ResponseBody`. Spring MVC використовує анотації на методах контролера для маршрутизації запитів: наприклад, `@GetMapping`, `@PostMapping`, `@PutMapping` позначають обробку HTTP GET, POST, PUT відповідно. Як наголошується в документації, «Spring MVC надає модель програмування на основі анотацій, де `@Controller` і `@RestController` використовують анотації для вираження мапінгів запитів, введення даних, обробки виключень та ін.». Для REST-контролера також діє правило, що повернений об'єкт перетворюється безпосередньо в JSON у відповіді (тобто дані кожного методу пишуться у тіло відповіді без рендерингу шаблону).

Наприклад, клас `AuthController` може обробляти запити автентифікації:

```
@RestController
@RequestMapping("/api/auth")
public class AuthController {
```

```
@Autowired
```

```

private AuthenticationService authService;

@PostMapping("/login")
public ResponseEntity<AuthResponse> authenticate(@RequestBody
AuthRequest request) {
    // Логіка аутентифікації через JWT
    AuthResponse response = authService.authenticateUser(request);
    return ResponseEntity.ok(response);
}

@PostMapping("/register")
public ResponseEntity<Client> register(@RequestBody Client client) {
    // Реєстрація нового клієнта
    Client created = authService.registerClient(client);
    return ResponseEntity.status(HttpStatus.CREATED).body(created);
}
}

```

Тут `@RestController` позначає, що контролер повертає дані (JSON) замість view. Шлях `@RequestMapping("/api/auth")` задає базовий маршрут, а методи `@PostMapping("/login")` та `/register` відповідають за обробку POST-запитів до `/api/auth/login` і `/api/auth/register`. Клас `AuthController` використовує сервіс `AuthenticationService` для виконання логіки входу та реєстрації, а на відповідь повертається статус 200 або 201 разом з об'єктом.

Інший приклад – `ClientController` для операцій з клієнтами:

```

@RestController
@RequestMapping("/api/clients")
public class ClientController {

    @Autowired

```

```

private ClientService clientService;

@GetMapping("/{id}")
public ResponseEntity<Client> getClient(@PathVariable Long id) {
    Client client = clientService.findById(id);
    return ResponseEntity.ok(client);
}

@PostMapping
public ResponseEntity<Client> createClient(@RequestBody Client client) {
    Client created = clientService.createClient(client);
    return ResponseEntity.status(HttpStatus.CREATED).body(created);
}

@PutMapping("/{id}")
public ResponseEntity<Client> updateClient(@PathVariable Long id,
                                           @RequestBody Client clientData) {
    Client updated = clientService.updateClient(id, clientData);
    return ResponseEntity.ok(updated);
}
}

```

У цьому прикладі методи контролера прив'язуються до URL з параметрами ({id}) та тіла запитів. Наприклад, `@GetMapping("/{id}")` віддає клієнта з заданим ідентифікатором, викликаючи сервіс. Атрибут `@PathVariable` зв'язує частину шляху з параметром методу, а `@RequestBody` десеріалізує JSON із тіла запиту в об'єкт. Такий підхід «розділяє стратегії»: контролери лише маршрутизують і викликають сервіси, не містять складної логіки.

3.5 Конфігурація безпеки (Spring Security, JWT, OAuth2)

Безпека в проєкті реалізована за допомогою Spring Security з токенами JWT та підтримкою OAuth2-логіну. Для початку, додано залежність `spring-boot-starter-security`, яка активує стандартний механізм Spring Security (фільтр-ланцюжок безпеки, базову конфігурацію). Для роботи з JWT використовується бібліотека `jjwt` (`io.jsonwebtoken`) – вона дозволяє створювати і перевіряти підписані JSON Web Token. Для OAuth2-доступу підключено `spring-security-oauth2-client`, що дає можливість конфігурувати клієнтів OAuth2 (наприклад, Google, GitHub).

Головний клас конфігурації безпеки `SecurityConfig` зазвичай містить анотацію `@EnableWebSecurity` і перевизначає налаштування `HttpSecurity`. Наприклад:

```
@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private JwtAuthenticationFilter jwtFilter;

    @Autowired
    private OAuth2LoginSuccessHandler oAuth2SuccessHandler;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .cors().and().csrf().disable()           // вимикаємо CSRF, вмикаємо CORS
            .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS) // без
сесій
            .and()
            .authorizeRequests()
```

```

        .antMatchers("/api/auth/**", "/oauth2/**").permitAll() // відкриті
// шляхи

        .anyRequest().authenticated() // всі інші потребують аутентифікації
        .and()
        .oauth2Login()
        .successHandler(oAuth2SuccessHandler) // обробник при успішному
OAuth2-логіні
        .and()
        .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);
// JWT-фільтр перед стандартною перевіркою
    }
}

```

У цій конфігурації ми вимикаємо CSRF (оскільки маємо stateless API), встановлюємо політику STATELESS, тобто кожен запит аутентифікується незалежно. У `authorizeRequests()` вказано, що шляхи `/api/auth/**` та `/oauth2/**` відкриті (`permitAll`), а всі інші потребують авторизації. Далі налаштовується OAuth2-вхід через `.oauth2Login()` з власним `successHandler`. І врешті додається `JwtAuthenticationFilter` через `addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class)`, щоб перехопити запити перед стандартним фільтром Spring Security.

`JwtAuthenticationFilter` – це кастомний фільтр, що наслідує, наприклад, `OncePerRequestFilter`. У ньому зазвичай береться заголовок `Authorization: Bearer <token>`, парситься токен (з використанням `jjwt`), перевіряється підпис і термін дії. Якщо токен валідний, у контекст безпеки встановлюється `UsernamePasswordAuthenticationToken` з інформацією про користувача. Таким чином, кожен захищений запит спочатку проходить через цей фільтр, що відповідає за JWT-автентифікацію.

`OAuth2LoginSuccessHandler` – це клас, що реалізує `AuthenticationSuccessHandler`. Його метод `onAuthenticationSuccess` викликається після успішного логіну через OAuth2 (напр., Google). У проєкті цей обробник може

створювати JWT на основі отриманого OAuth2-аутентифікованого користувача і повертати його у відповідь, аби клієнт використав токен далі. Наприклад, після входу за OAuth2 можна видати власний JWT і перенаправити клієнта або відправити токен у JSON-відповіді. Інтерфейс AuthenticationSuccessHandler описує «стратегію, що використовується для обробки успішної аутентифікації» (типово — перенаправлення або інша логіка).

Приклад простого SuccessHandler може виглядати так:

```
public class OAuth2LoginSuccessHandler implements
AuthenticationSuccessHandler {
    @Autowired
    private JwtTokenUtil jwtTokenUtil;

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request,
    HttpServletResponse response,
        Authentication authentication) throws IOException {
        // Генерувати JWT на основі authenticated користувача
        String token = jwtTokenUtil.generateToken(authentication.getName());
        response.getWriter().write("{\"token\":\"" + token + "\"}");
        response.setStatus(HttpServletResponse.SC_OK);
    }
}
```

Отже, механізм роботи безпеки такий: при вході за OAuth2 користувач перенаправляється на провайдера (Google/GitHub), після чого в разі успіху спрацьовує наш successHandler, що видає власний JWT. Далі клієнт прикріплює цей JWT у заголовок Authorization: Bearer <token> до кожного запиту. Усі такі запити перехоплює JwtAuthenticationFilter, перевіряючи токен і надаючи доступ до захищених ресурсів.

Офіційна документація Spring Security підтримує опис такого сценарію: наприклад, OAuth2 Login дозволяє користувачу виконувати вхід через існуючі облікові записи провайдерів (Google, GitHub). А Spring Security на стороні сервера при наявності заголовка Authorization: Bearer автоматично обробляє JWT за специфікацією OAuth2 Resource Server. Зокрема, наведено, що при наявності заголовка “Bearer” Spring перевіряє підпис і дійсність токена. Таким чином, в нашому проєкті реалізовано безстрокову аутентифікацію через JWT та конфігурацію OAuth2-клієнта, а використовувані бібліотеки (spring-boot-starter-security, jjwt, spring-security-oauth2-client) забезпечують необхідний функціонал безпеки.

ТЕСТУВАННЯ REST API

У даному розділі представлено тестування реалізованих REST API-ендпоінтів, які забезпечують доступ до функціоналу інформаційної системи обліку автосервісу.

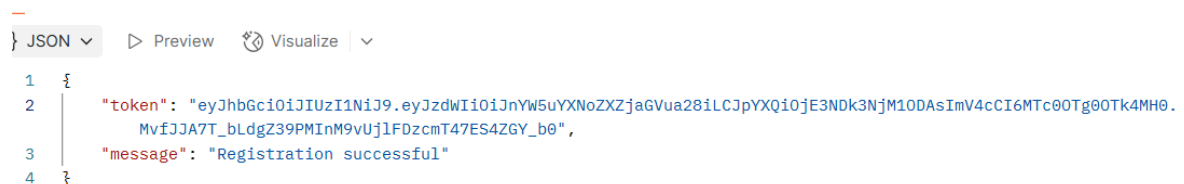
4.1 Тестування автентифікації користувача

Реєстрація нового клієнта (рис. 4.1).

- ендпоінт: POST /api/auth/register;
- опис: Створює новий обліковий запис клієнта.

Body:

```
{  
  "name": "Ганна Шевченко",  
  "email": "ganna.shevchenko@example.com",  
  "phone": "+380523434567",  
  "username": "gannashevchenko",  
  "password": "strongpassword"  
}
```



```
{ JSON ▾ ▷ Preview 🔄 Visualize ▾  
1  {  
2    "token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJnYW5uYXNoZXZjaGVua28iLCJpYXQiOiJlE3NDk3NjM1ODAsImV4cCI6MTc0OTg0Tk4MH0.  
    MvffJJA7T_bLdgZ39PMInM9vUj1FDzcmT47ES4ZGY_b0",  
3    "message": "Registration successful"  
4  }
```

Рисунок 4.1 – Успішна реєстрація нового клієнта

Автентифікація (отримання JWT) (рис. 4.2).

- ендпоінт: POST /api/auth/authenticate;
- опис: Повертає JWT-токен для подальших запитів.

Body:

```
{
```

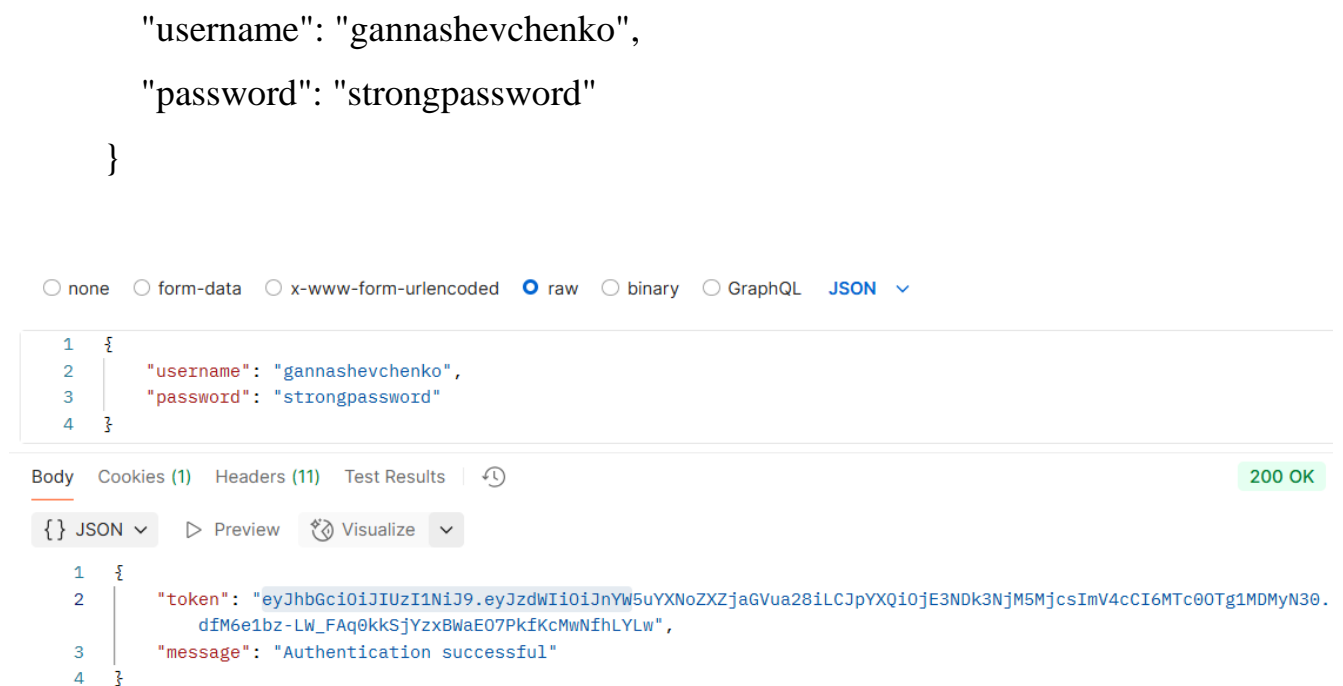
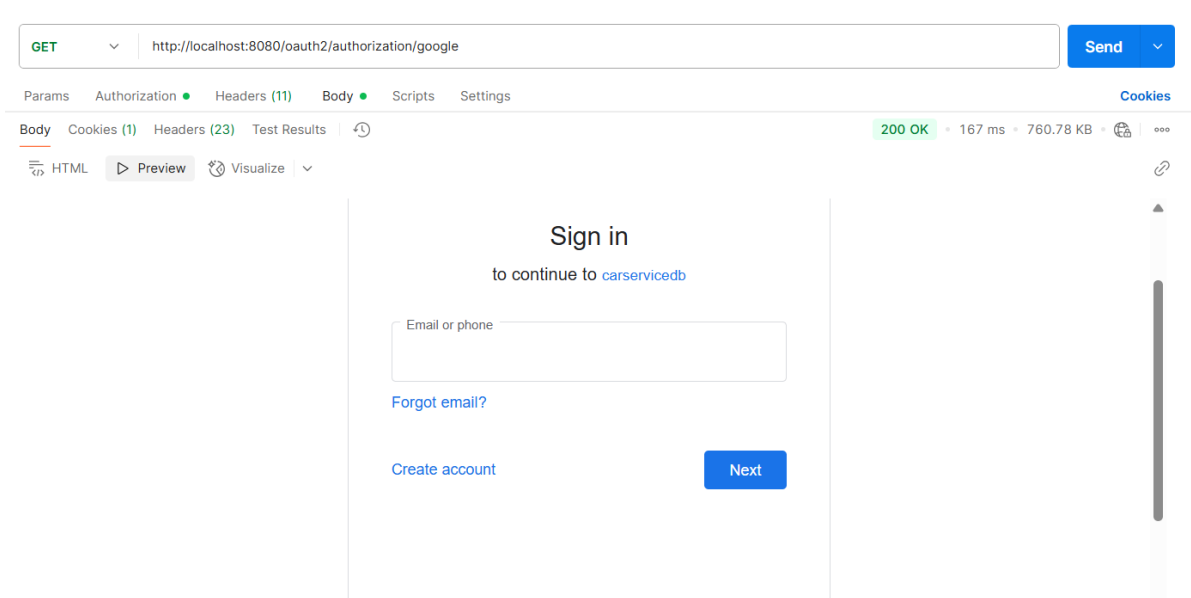


Рисунок 4.2 – Успішна автентифікація

Автентифікація OAuth2 (реєстрація/вхід) (рис. 4.3-4.8).

- ендпоінт для ініціації: GET <http://localhost:8080/oauth2/authorization/google>
- опис: Дозволяє користувачам входити в систему за допомогою облікового запису Google. Процес відбувається через веб-браузер, включаючи перенаправлення до Google для автентифікації та згоди на доступ, а потім повернення до додатку. При успішному вході, CustomOAuth2UserService створює або оновлює дані клієнта в базі даних на основі інформації з Google.



Создать аккаунт Google

Введите свое имя

Имя
Дмитро

Фамилия (необязательно)
Черненко

Далее

Выберите адрес Gmail

Выберите адрес Gmail или создайте свой собственный

☐ dmtcrnk@gmail.com

☐ dmitroc210@gmail.com

☒ Создать свой собственный адрес Gmail

Создать адрес Gmail

dmitro.chernenko123 @gmail.com

Можно использовать буквы, цифры и точки

Использовать существующий электронный адрес

Далі

Вхід через Google

Увійдіть у додаток carservicedb

dmitro.chernenko123@gmail.com

Продовжуючи, ви дозволяєте Google надати сервісу carservicedb ваші ім'я, електронну адресу й зображення профілю. Перегляньте Політику конфіденційності й Умови використання сервісу carservicedb.

Ви можете керувати функцією "Вхід через Google" у своєму обліковому записі Google.

Скасувати Продовжити

```
"token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWUiOiJkbWl0cm8uY2hlcm5lbmtvIiwiaWF0IjoxNzQ5NzY4NzAxLCJleHAiOiJE3NDk4NTUxMDF9.mCH-yqHwe9ENIwaUy0t2QVcS3i5I1Ltzg3t0ixarfw0",  
"message": "Authentication successful"
```

Рисунки 4.3-4.7 – Успішна реєстрація та автентифікація OAuth2

4.2 Тестування клієнтських операцій

Створення нового клієнта (рис. 4.8).

Ендпоінт: POST <http://localhost:8080/api/clients>

Body:

```
{  
  "name": "Іван Сидоренко",  
  "email": "ivan.syndorenko@example.com",  
  "phone": "+380508934777",  
  "username": "ivansydorenko",
```

```

    "password": "strongpassword",
    "role": "USER"
}

```

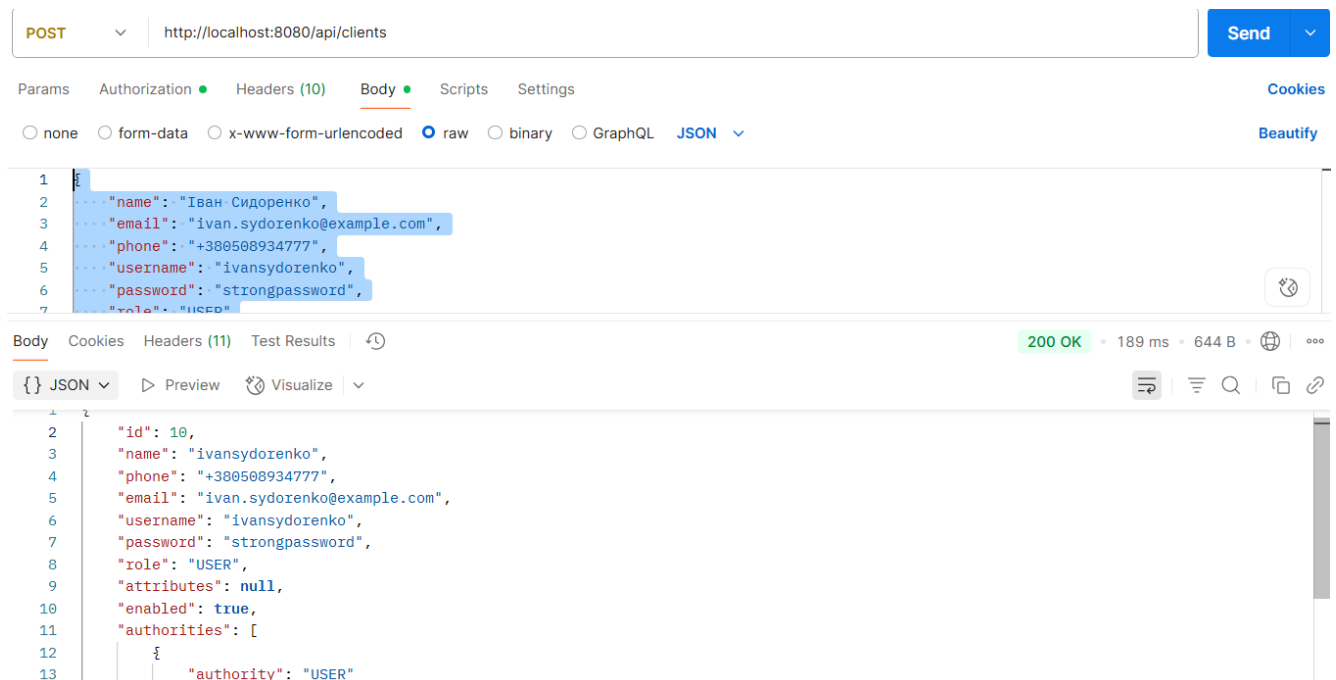


Рисунок 4.8 – Створення нового клієнта

Отримання списку клієнтів (рис. 4.9).

Ендпоінт: GET `http://localhost:8080/api/clients`

	id	name	phone	email	username	password
0	1	ivan_petrov	+380991234567	ivan.petrov@example.com	ivan_petrov	\$2a\$10\$Au.rQFVH7r6.HFi4HKpHDOK4GMsDDmnut60ZQ/bcRu74.ykJ41ulq
1	2	irasydorenko	+380501234567	irina.sydoenko@example.com	irasydorenko	\$2a\$10\$QVfnmRm/YQpGhmEpP.cT4eczl7tSx4Xcb5DkylEmny4TtPr1eYvmu
2	4	dmitro.chernenko	+380765434567	dmitro.chernenko123@gmail.com	dmitro.chernenko	\$2a\$10\$ZY.BMIZhz3eReb3rcVQcy.WfuX4h.ODgCDJyVq4LBLhf/N0Qyogdq
3	9	andriyshevchenko	+380523434590	andriy.shevchenko@example.com	andriyshevchenko	\$2a\$10\$VrCuFi1HB13GvsrlHAKh.Im6aeRxW961apyETJpJNyLqDH99Paik
4	10	ivansydorenko	+380508934777	ivan.sydoenko@example.com	ivansydorenko	strongpassword
5	11	gannashevchenko	+380523434567	ganna.shevchenko@example.com	gannashevchenko	strongpassword

Рисунок 4.9 – Отримання списку клієнтів

Оновлення клієнта (рис. 4.10).

Ендпоінт: PUT <http://localhost:8080/api/clients/11>

Body:

```
{
  "name": "gannashevchenko",
  "phone": "+380971110000",
  "email": "anutashevchenko@example.com",
  "username": "gannashevchenko",
  "password": "strongpassword",
  "role": "USER"
}
```

1	2	irasydorenko	+380501234567	irina.sydorenko@example.com	irasydorenko
2	4	dmitro.chernenko	+380765434567	dmitro.chernenko123@gmail.com	dmitro.chernenko
3	9	andriyshevchenko	+380523434590	andriy.shevchenko@example.com	andriyshevchenko
4	10	ivansydorenko	+380508934777	ivan.sydorenko@example.com	ivansydorenko
5	11	gannashevchenko	+380523434567	ganna.shevchenko@example.com	gannashevchenko

id	11
name	gannashevchenko
phone	+380971110000
email	anutashevchenko@example.com
username	gannashevchenko
password	strongpassword
role	USER

Рисунок 4.10 – Оновлення номера та пошти клієнта з id 3

Видалення клієнта (рис. 4.11).

Ендпоінт: DELETE <http://localhost:8080/api/clients/11>

	id	name	phone	email	username
0	1	ivan_petrov	+380991234567	ivan.petrov@example.com	ivan_petrov
1	2	irasydorenko	+380501234567	irina.sydorenko@example.com	irasydorenko
2	4	dmitro.chernenko	+380765434567	dmitro.chernenko123@gmail.com	dmitro.chernenko
3	9	andriyshevchenko	+380523434590	andriy.shevchenko@example.com	andriyshevchenko
4	10	ivansydorenko	+380508934777	ivan.sydorenko@example.com	ivansydorenko

Рисунок 4.11 – Список клієнтів після видалення клієнта з id 11

4.3 Тестування автомобілів клієнтів

Додавання нового авто клієнту (рис. 4.12).

Ендпоінт: POST <http://localhost:8080/api/cars>

Body:

```
{
  "client": {
    "id": 1
  },
  "make": "Toyota",
  "model": "Camry",
  "year": 2020,
  "vin": "VIN1234567890ABCDEF"
}
```

Body	Cookies	Headers (11)	Test Results
{ } JSON Preview Visualize			
id	3		
> client	{13}		
make	Toyota		
model	Camry		
year	2020		
vin	VIN1234567890ABCDEF		

Рисунок 4.12 – Додавання нового авто клієнту з id 1

Отримання авто клієнта (рис. 4.13).

Ендпоінт: GET <http://localhost:8080/api/cars/client/1>

id	client		make	model	year	vin
3	id	1	Toyota	Camry	2020	VIN1234567890ABCDEF
	name	ivan_petrov				
	phone	+380991234567				
	email	ivan.petrov@example.com				
	username	ivan_petrov				
	password	\$2a\$10\$Au.rQFVH7r6.HFi4HKpHDOK4GMSDDmnut60ZQ/bcRu74.ykJ41ulq				

Рисунок 4.13 – Отримання авто клієнта з id 1

Оновлення авто (рис. 4.14).

Ендпоінт: PUT <http://localhost:8080/api/cars/3>

Body:

```
{
  "make": "Honda",
  "model": "Civic",
  "year": 2018,
  "vin": "NEWVINABCDEF1234567"
}
```

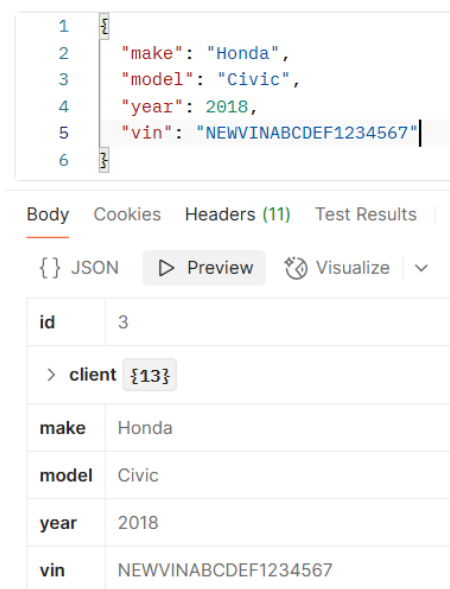


Рисунок 4.14 – Оновлення авто з id 3

Видалення авто (рис. 4.15).

Ендпоінт: DELETE http://localhost:8080/api/cars/3

	id	client	make	model	year	vin
0	2	<input type="text" value="null"/>	Toyota	Camry	2018	AA1234BB
1	4	id	BMW	X5	2019	WBAKS8C5XFD888001
		name				
		phone				
		email				
		username				
		password				
		role				
		attributes				
		enabled				
		authorities				
2	5	id	Tesla	Model 3	2021	5YJ3E1EA7LF000321
		name				
		phone				
		email				
		username				
		password				
		role				
		attributes				
		enabled				
		authorities				
3	6	id	Volkswagen	Golf	2018	WVWA7AU5XFW111222
		name				
		phone				
		email				
		username				
		password				
		role				
		attributes				
		enabled				
		authorities				
4	7	id	Mercedes-Benz	GLC	2022	WDC0G4JB1LF999888
		name				
		phone				
		email				
		username				
		password				
		role				
		attributes				
		enabled				
		authorities				

Рисунок 4.15 – Список машин після видалення машини з id 3

4.4 Тестування операцій з механіками

Додавання нового майстра (рис. 4.16).

Ендпоінт: POST <http://localhost:8080/api/mechanics>

Body:

```
{  
  "name": "Іван Ванюк",  
  "specialization": "Двигуни"  
}
```

id	2
name	Іван Ванюк
specialization	Двигуни

Рисунок 4.16 – Додавання нового майстра

Отримання списку майстрів (рис. 4.17).

Ендпоінт: GET <http://localhost:8080/api/mechanics>

	id	name	specialization
0	2	Іван Ванюк	Двигуни
1	3	Олексій Ткаченко	Гальмівна система
2	4	Сергій Чернявський	Підвіска і рульове керування
3	5	Наталія Ковальчук	Двигуни

Рисунок 4.17 – Отримання списку майстрів

Оновлення майстра (рис. 4.18).

Ендпоінт: PUT <http://localhost:8080/api/mechanics/2>

Body:

```
{  
  "name": "Іван Цимбалюк",  
  "specialization": "Ходова"  
}
```

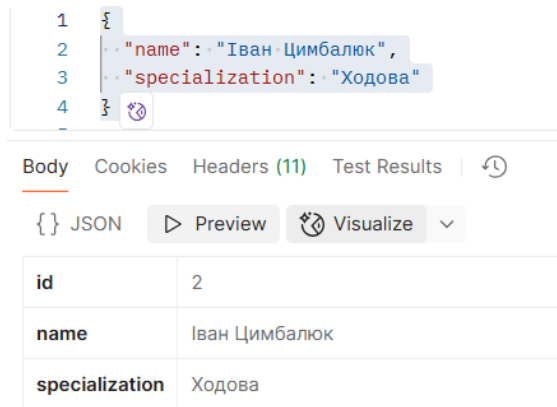


Рисунок 4.18 – Оновлення майстра з id 2

Видалення майстра (рис. 4.19).

Ендпоінт: DELETE <http://localhost:8080/api/mechanics/2>

	id	name	specialization
0	3	Олексій Ткаченко	Гальмівна система
1	4	Сергій Чернявський	Підвіска і рульове керування
2	5	Наталія Ковальчук	Двигуни

Рисунок 4.19 – Список майстрів після видалення майстра з id 2

4.5 Тестування записів на обслуговування

Створення запису на обслуговування (рис. 4.20).

Ендпоінт: POST <http://localhost:8080/api/records>

Body:

```

{
  "car": {
    "id": 7
  },
  "mechanic": {
    "id": 5
  },
  "date": "2025-06-15",

```

```
"description": "Заміна масла та фільтрів"
}
```

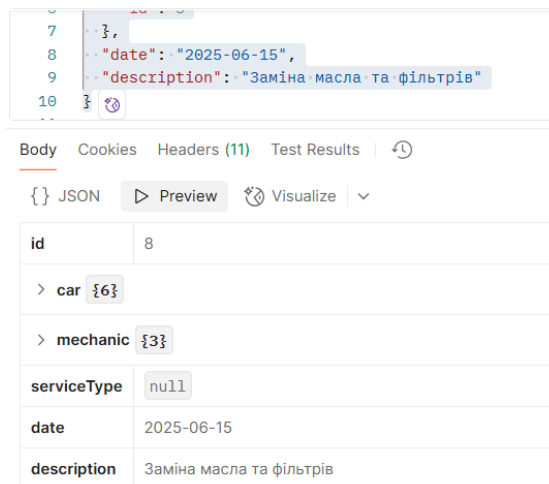


Рисунок 4.20 – Створення запису на обслуговування

Отримання записів обслуговування автомобіля (рис. 4.21).

Ендпоінт: GET <http://localhost:8080/api/records>

id	car		mechanic		serviceType	date	descripti			
	id		id							
8	client	id	10	name	Наталія Ковальчук	null	2025-06-15	Заміна масла та фільтрів		
									specialization	Двигуни
9	client	id	2	name	Ірина Ковальчук	null	2025-06-15	Заміна масла та фільтрів		
									specialization	Двигуни

Рисунок 4.21 – Отримання списку записів обслуговування автомобіля (на
рисунку не повний список)

Отримання записів обслуговування майстра (рис. 4.22).

Ендпоінт: GET <http://localhost:8080/api/records/mechanic/5>

GET

http://localhost:8080/api/records/mechanic/5

Send

Params

Authorization

Headers (10)

Body

Scripts

Settings

Cookies

Body

Cookies

Headers (11)

Test Results

200 OK

462 ms

2.24 KB

{ } JSON

Preview

Visualize

	id	car	mechanic		serviceType	date	descrip
8	id	7	id	5	null	2025-06-15	Заміна масла 1 фільтрі
	client		name	Наталія Ковальчук			
		id	specialization	Двигуни			
		name					
		phone					
		email					
		username					
		password					
		role					

9	id	4	id	5	null	2025-06-15	Заміна масла 1 фільтрі
	client		name	Наталія Ковальчук			
		id	specialization	Двигуни			
		name					
		phone					
		email					
		username					
		password					
		role					
		attributes					
1		enabled					

id	4	id	5	null	2025-06-25	Перевірка та заміна зношених елементів підвіски
client		name	Наталія Ковальчук			
	id	specialization	Двигуни			
	name					
	phone					
	email					
	username					
	password					
	role					
	attributes					

Рисунок 4.22 – Отримання списку записів обслуговування майстра з id 5

Оновлення запису (рис. 4.23).

Ендпоінт: PUT <http://localhost:8080/api/records/8>

Body:

```
{
  "car": {
```

```

    "id": 7
  },
  "mechanic": {
    "id": 5
  },
  "date": "2025-06-22",
  "description": "Регулювання розвалу-сходження"
}

```

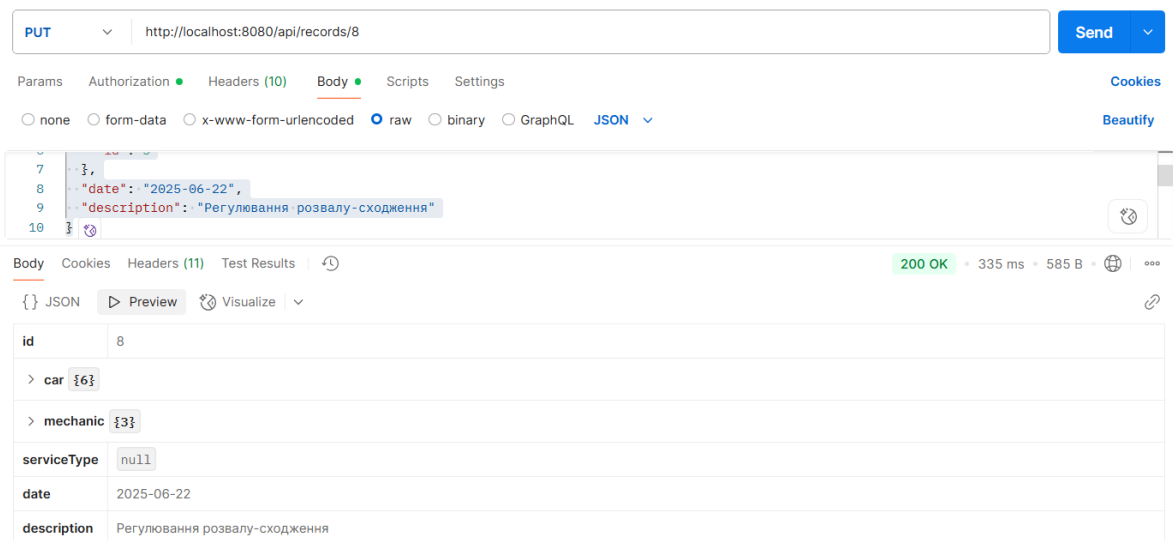


Рисунок 4.23 – Оновлення запису з id 8

Видалення запису запису (рис. 4.24).

Ендпоінт: DELETE `http://localhost:8080/api/records/8`

id	car		mechanic		serviceType	date	description
9	id	4	id	5	null	2025-06-15	Заміна масла та фільтрів
	client		name	Наталія Ковальчук			
	id	2	specialization	Двигуни			
		name	irasydorenko				
		phone	+380501234567				
10	id	5	id	4	null	2025-06-20	Заміна передніх задніх гальмівні колодок
	client		name	Сергій Чернявський			
	id	4	specialization	Підвіска і рульове керування			
		name	dmitro.chernenko				
		phone	+380765434567				
		email	dmitro.chernenko123@gmail.com				

Рисунок 4.24 – Список записів після видалення запису з id 8 (не повний)

4.6 Тестування типів послуг

Додавання типу послуги (рис. 4.25).

Ендпоінт: POST <http://localhost:8080/api/types>

Body:

```
{  
  "name": "Заміна масла та фільтрів",  
  "standardPrice": 500.00  
}
```

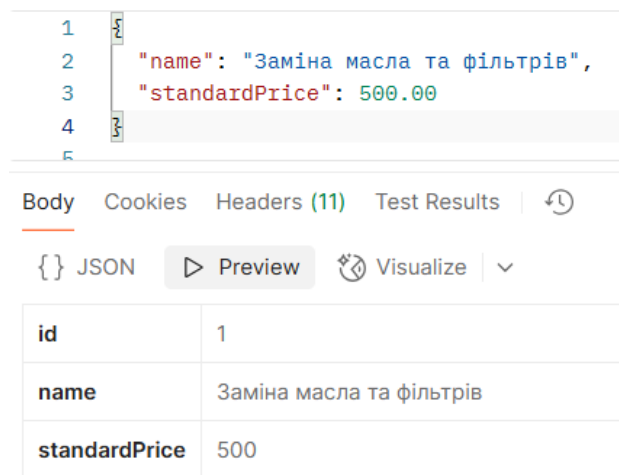


Рисунок 4.25 – Додавання типу послуги

Отримання всіх типів послуг (рис. 4.26).

Ендпоінт: GET <http://localhost:8080/api/types>

	id	name	standardPrice
0	1	Заміна масла та фільтрів	500
1	2	Заміна передніх і задніх гальмівних колодок	700
2	3	Регулювання розвалу-сходження	400
3	4	Перевірка та заміна зношених елементів підвіски	900

Рисунок 4.26 – Список всіх типів послуг

Призначення типу послуги запису (рис. 4.27).

Ендпоінт: PUT <http://localhost:8080/api/records/9/assign-service-type/1>

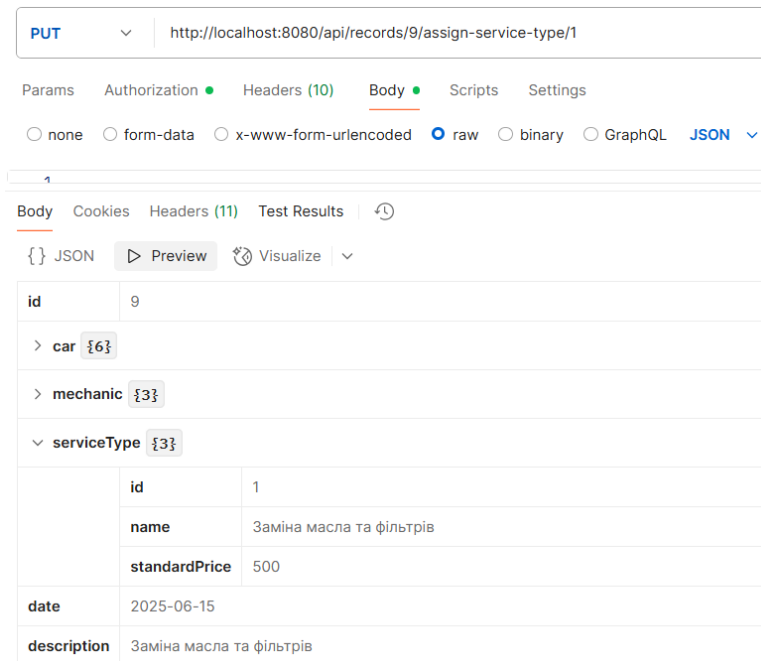


Рисунок 4.27 – Призначення типу послуги з id 1 запису з id 9

4.7 Статистика та розширені запити

Отримання загальної суми обслуговування автомобіля (рис. 4.28).

Ендпоінт: GET <http://localhost:8080/api/records/car/4/total-cost>

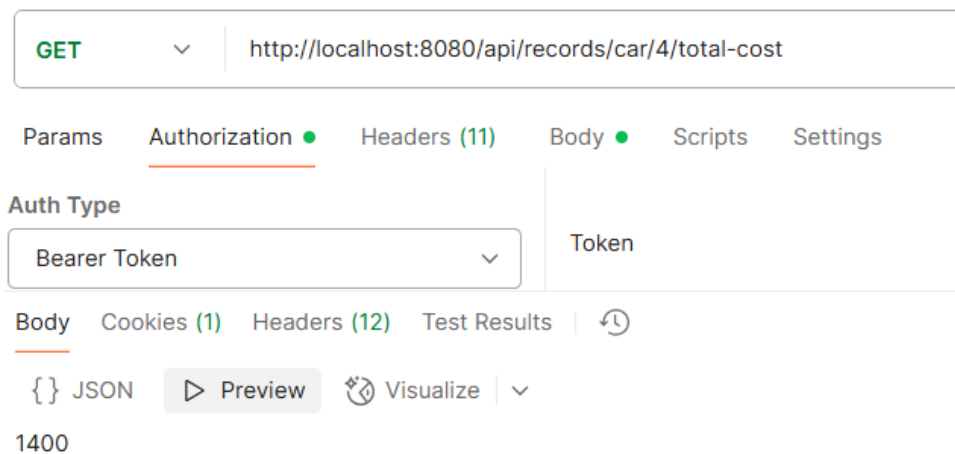


Рисунок 4.28 – Отримання загальної суми обслуговування автомобіля з id 4

Отримання статистики роботи майстра (кількість записів) (рис. 4.29).

Ендпоінт: GET <http://localhost:8080/api/records/mechanic/5/count>

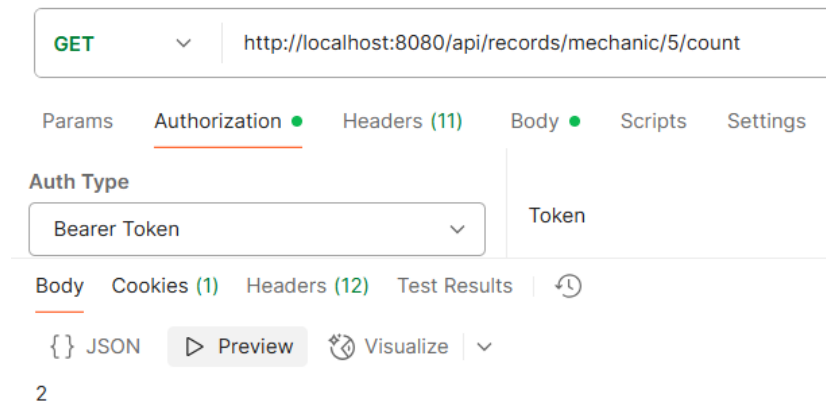


Рисунок 4.29 – Отримання статистики роботи майстра з id 5

Отримання найпопулярніших послуг (рис. 4.30).

Ендпоінт: GET http://localhost:8080/api/records/popular-service-types

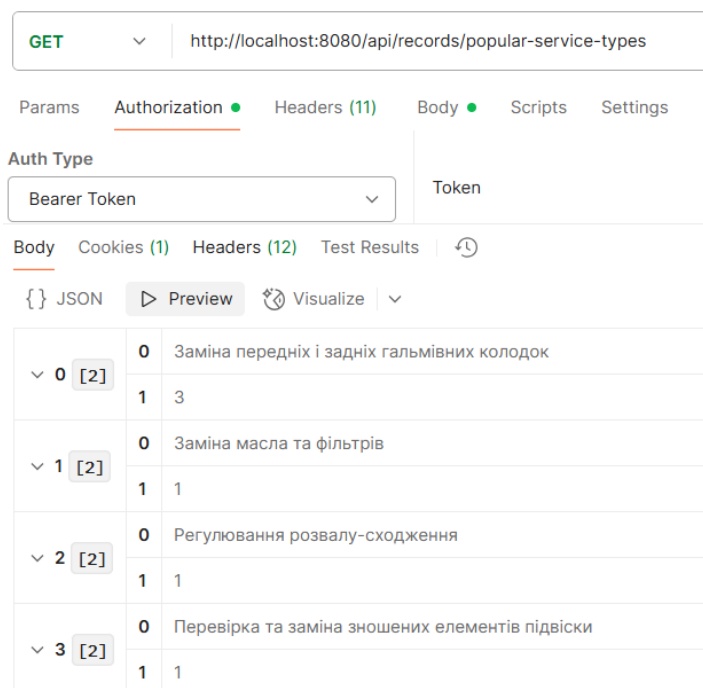


Рисунок 4.30 – Отримання найпопулярніших послуг (рейтинг)

Отримання обслуговування за період (рис. 4.31).

Ендпоінт: GET http://localhost:8080/api/records/between?start=2025-06-15&end=2025-06-20

GET

http://localhost:8080/api/records/between?start=2025-06-15&end=2025-06-20

Send

ParamsAuthorizationHeaders (11)BodyScriptsSettings

BodyCookies (1)Headers (12)Test Results

200 OK597 ms3.6 KB

{ }JSONPreviewVisualize

car		mechanic		serviceType		date
id	4	id	5	id	1	2025-06-15
client		name	Наталія Ковальчук	name	Заміна масла та фільтрів	
		specialization	Двигуни	standardPrice	500	
		id	2			
		name	irasydorenko			
		phone	+380501234567			
		email	irina.sydorenko@example.com			
		username	irasydorenko			
id	5	id	4	id	2	2025-06-20
client		name	Сергій Чернявський	name	Заміна передніх і задніх гальмівних колодок	
		specialization	Підвіска і рульове керування	standardPrice	700	
		id	4			
		name	dmitro.chernenko			
		phone	+380765434567			
		email	dmitro.chernenko123@gmail.com			
		username	dmitro.chernenko			
id	6	id	4	id	2	2025-06-20
client		name	Сергій Чернявський	name	Заміна передніх і задніх гальмівних колодок	
		specialization	Підвіска і рульове керування	standardPrice	700	
		id	9			
		name	andriyshevchenko			
		phone	+380523434590			
		email	andriy.shevchenko@example.com			
		username	andriyshevchenko			
id	7	id	4	id	2	2025-06-20
client		name	Сергій Чернявський	name	Заміна передніх і задніх гальмівних колодок	
		specialization	Підвіска і рульове керування	standardPrice	700	
		id	10			
		name	ivansydorenko			
		phone	+380508934777			
		email	ivan.sydorenko@example.com			
		username	ivansydorenko			

Рисунок 4.31 – Список обслуговувань за період з 2025-06-15 по 2025-06-20

Отримання автомобілей з найчастішим обслуговуванням (рис. 4.32).

Ендпоінт: GET http://localhost:8080/api/records/top-cars

GET

▼

http://localhost:8080/api/records/top-cars

Params

Authorization ●

Headers (11)

Body ●

Scripts

Body

Cookies (1)

Headers (12)

Test Results

🕒

{ }

JSON

▶ Preview

🔗 Visualize

▼

▼ 0 [2]	0	4
	1	2
▼ 1 [2]	0	6
	1	2
▼ 2 [2]	0	5
	1	1
▼ 3 [2]	0	7
	1	1

Рисунок 4.32 – Отримання автомобілей з найчастішим обслуговуванням (рейтинг)

ЗАГАЛЬНІ ВИСНОВКИ

У процесі виконання курсової роботи було повністю реалізовано серверний вебзастосунок для управління обліком клієнтів, автомобілів, сервісних послуг, механіків та записів технічного обслуговування. Проєкт відповідає сучасним вимогам до архітектури бекенд-систем та демонструє вміння застосовувати на практиці принципи об'єктно-орієнтованого програмування.

Вивчення предметної області дало змогу чітко сформулювати вимоги до інформаційної системи, визначити ключові сутності, зв'язки між ними та побудувати ефективну логічну модель. Було виявлено основні проблеми, що характерні для автосервісів, які працюють без цифрової системи обліку: дублювання даних, ручне введення, труднощі з пошуком інформації, відсутність прозорості історії обслуговувань тощо. Запропоноване рішення дозволяє автоматизувати ці процеси, підвищуючи загальну ефективність та безпеку даних.

З технічної точки зору, застосунок реалізовано за допомогою мови Java з використанням фреймворку Spring Boot. Архітектура побудована за моделлю Controller – Service – Repository, що забезпечує чітке розділення обов'язків, підтримуваність і масштабованість проєкту. Всі сутності системи реалізовано через JPA-entity з відповідними зв'язками (1:N), що дозволяє ефективно зберігати інформацію у реляційній базі даних.

Для взаємодії з клієнтом реалізовано повноцінний REST API, який підтримує всі CRUD-операції для сутностей Client, Car, Mechanic, ServiceType та ServiceRecord. Забезпечено підтримку фільтрації, пошуку, а також реалізовано додаткові аналітичні запити (наприклад, статистика по майстрах, найбільш обслуговувані автомобілі, популярні види послуг). Таке розширення функціоналу дозволяє керівництву сервісного центру приймати більш обґрунтовані рішення.

Одним із важливих компонентів розробки стала система безпеки. Для автентифікації та авторизації реалізовано два механізми: логін/пароль з JWT-токенами та OAuth2 через Google. Це дає змогу забезпечити гнучкий та захищений доступ до API для різних категорій користувачів: адміністраторів, клієнтів і

сервісних інженерів. Система ролей гарантує обмеження доступу до ресурсів відповідно до прав користувача.

Після завершення розробки всі функціональні можливості системи було протестовано за допомогою Postman. Це дозволило виявити і виправити помилки, перевірити логіку обробки запитів, а також впевнитися в коректній роботі всіх компонентів. Тестування підтвердило, що API відповідає принципам REST та забезпечує зручність у використанні з фронтенд-застосунками або мобільними додатками.

У результаті виконання роботи сформовано низку важливих практичних навичок, зокрема: робота з фреймворком Spring Boot, побудова безпечного REST API, налаштування JWT та OAuth2, тестування вебсервісів, проєктування об'єктної моделі, використання JPA та бази даних. Крім того, опрацьовано й теоретичну складову, зокрема принципи SOLID, розділення відповідальностей, інверсію управління, інкапсуляцію та повторне використання коду.

Розроблена система може бути основою для повноцінного комерційного рішення, оскільки задовольняє базові вимоги до систем обліку технічного обслуговування автотранспорту. Вона може бути легко масштабована, доповнена вебінтерфейсом або мобільним додатком. В перспективі можливе розширення функціоналу системи аналітикою в реальному часі, інтеграцією з системами оплати або CRM.

Таким чином, мета курсової роботи досягнута повною мірою. Результати виконання завдання демонструють високий рівень володіння сучасними технологіями backend-розробки, а отримані знання та досвід стануть міцною основою для подальшої фахової підготовки в галузі комп'ютерних наук.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. The Role of Digitalization in the Future of Car Repair. URL: <https://fixico.com/blog/the-role-of-digitalization-in-the-future-of-car-repair> (дата звернення: 03.06.2025).
2. Workshop Management Software Includes Automated Billing Features – 5 Common Challenges in Auto Repair. URL: <https://cogxim.medium.com/workshop-management-software-includes-automated-billing-features5-common-challenges-in-auto-ae2426efcaad> (дата звернення: 03.06.2025).
3. Accessing Data with JPA. Spring Guides. URL: <https://spring.io/guides/gs/accessing-data-jpa> (дата звернення: 05.06.2025).
4. Spring Data JPA – Reference Documentation. URL: <https://docs.spring.io/spring-data/jpa/docs/current-SNAPSHOT/reference/html/#reference> (дата звернення: 09.06.2025).
5. Fowler M. Service Layer [Electronic resource] // MartinFowler.com. URL: <https://martinfowler.com/eaCatalog/serviceLayer.html> (дата звернення: 09.06.2025).

ДОДАТОКИ

Код програми

```
package ua.opnu.practice1kr.Auth;

import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.bind.annotation.*;
import ua.opnu.practice1kr.Auth.Dto.AuthenticationRequest;
import ua.opnu.practice1kr.Auth.Dto.AuthenticationResponse;
import ua.opnu.practice1kr.Auth.Dto.RegisterRequest;
import ua.opnu.practice1kr.Client.Client;
import ua.opnu.practice1kr.Client.ClientService;
import ua.opnu.practice1kr.Jwt.JwtUtil;

@RestController
@RequestMapping("/api/auth")
@RequiredArgsConstructor
public class AuthController {

    private final ClientService clientService;
    private final JwtUtil jwtUtil;
    private final AuthenticationManager authenticationManager;

    @PostMapping("/register")
    public ResponseEntity<AuthenticationResponse> register(@RequestBody RegisterRequest
request) {
        Client client = Client.builder()
            .name(request.getName())
            .email(request.getEmail())
            .phone(request.getPhone())
            .username(request.getUsername())
            .password(request.getPassword())
            .build();
        Client registeredClient = clientService.registerNewClient(client);

        Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(
                request.getUsername(),
                request.getPassword()
            )
        );
        SecurityContextHolder.getContext().setAuthentication(authentication);
        String jwtToken = jwtUtil.generateToken(registeredClient);
        return
```



```

    ResponseEntity.ok(AuthenticationResponse.builder().token(jwtToken).message("Registration
successful").build());
}

@PostMapping("/authenticate")
public ResponseEntity<AuthenticationResponse> authenticate(@RequestBody
AuthenticationRequest request) {
    Authentication authentication = authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(
            request.getUsername(),
            request.getPassword()
        )
    );
    SecurityContextHolder.getContext().setAuthentication(authentication);
    Client authenticatedClient = (Client) authentication.getPrincipal();
    String jwtToken = jwtUtil.generateToken(authenticatedClient);
    return
    ResponseEntity.ok(AuthenticationResponse.builder().token(jwtToken).message("Authenticatio
n successful").build());
}

@GetMapping("/oauth2/success")
public ResponseEntity<AuthenticationResponse> oauth2LoginSuccess(Authentication
authentication) {
    if (authentication != null && authentication.getPrincipal() instanceof Client) {
        Client client = (Client) authentication.getPrincipal();
        String jwtToken = jwtUtil.generateToken(client);
        return
        ResponseEntity.ok(AuthenticationResponse.builder().token(jwtToken).message("OAuth2
Login successful").build());
    }
    return
    ResponseEntity.badRequest().body(AuthenticationResponse.builder().message("OAuth2 login
failed").build());
}
}

```

```

package ua.opnu.practice1kr.Auth.Dto;

```

```

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

```

```

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class AuthenticationRequest {
    private String username;
}

```

```

        private String password;
    }

package ua.opnu.practice1kr.Auth.Dto;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class AuthenticationResponse {
    private String token;
    private String message;
}

```

```

package ua.opnu.practice1kr.Auth.Dto;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class RegisterRequest {
    private String name;
    private String email;
    private String phone;
    private String username;
    private String password;
}

```

```

package ua.opnu.practice1kr.Car;

import jakarta.persistence.*;
import lombok.*;
import ua.opnu.practice1kr.Client.Client;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder

```

```

public class Car {

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

    @ManyToOne
    @JoinColumn(name = "client_id")
    private Client client;

    private String make;

    private String model;

    private Integer year;

    private String vin;
}

package ua.opnu.practice1kr.Car;

import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/cars")
@RequiredArgsConstructor
public class CarController {

    private final CarService carService;

    @GetMapping
    public List<Car> getAll() {
        return carService.getAll();
    }

    @GetMapping("/{id}")
    public Car getById(@PathVariable Long id) {
        return carService.getById(id);
    }

    @GetMapping("/client/{clientId}")
    public List<Car> getByClient(@PathVariable Long clientId) {
        return carService.getByClientId(clientId);
    }

    @PostMapping
    public Car create(@RequestBody Car car) {
        return carService.save(car);
    }
}

```

```

    }

    @PutMapping("/{id}")
    public Car update(@PathVariable Long id, @RequestBody Car car) {
        return carService.update(id, car);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        carService.delete(id);
    }
}

```

```

package ua.opnu.practice1kr.Car;

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

import java.util.List;

public interface CarRepository extends JpaRepository<Car, Long> {
    List<Car> findByClientId(Long clientId);
}

```

```

package ua.opnu.practice1kr.Car;

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
@RequiredArgsConstructor
public class CarService {

    private final CarRepository carRepository;

    public List<Car> getAll() {
        return carRepository.findAll();
    }

    public Car getById(Long id) {
        return carRepository.findById(id).orElseThrow();
    }

    public List<Car> getByClientId(Long clientId) {
        return carRepository.findByClientId(clientId);
    }

    public Car save(Car car) {

```

```

        return carRepository.save(car);
    }

    public Car update(Long id, Car updatedCar) {
        Car car = getById(id);
        car.setMake(updatedCar.getMake());
        car.setModel(updatedCar.getModel());
        car.setYear(updatedCar.getYear());
        car.setVin(updatedCar.getVin());
        return carRepository.save(car);
    }

    public void delete(Long id) {
        carRepository.deleteById(id);
    }
}

```

```

package ua.opnu.practice1kr.Client;

```

```

import jakarta.persistence.*;
import lombok.*;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.oauth2.core.user.OAuth2User;
import ua.opnu.practice1kr.Client.Role.Role;

```

```

import java.util.Collection;
import java.util.List;
import java.util.Map;

```

```
@Entity
```

```
@Data
```

```
@NoArgsConstructor
```

```
@AllArgsConstructor
```

```
@Builder
```

```
public class Client implements UserDetails, OAuth2User {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    private String name;
```

```
    private String phone;
```

```
    @Column(unique = true)
```

```
    private String email;
```

```
    @Column(unique = true, nullable = false)
```

```
    private String username;
```

```

@Column(nullable = false)
private String password;

@Builder.Default
@Enumerated(EnumType.STRING)
private Role role = Role.USER;

@Transient
private Map<String, Object> attributes;

@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    return List.of(new SimpleGrantedAuthority(role.name()));
}

@Override
public boolean isAccountNonExpired() {
    return true;
}

@Override
public boolean isAccountNonLocked() {
    return true;
}

@Override
public boolean isCredentialsNonExpired() {
    return true;
}

@Override
public boolean isEnabled() {
    return true;
}

@Override
public Map<String, Object> getAttributes() {
    return attributes;
}

@Override
public String getName() {
    return this.username;
}
}

```

```

package ua.opnu.practice1kr.Client;

```

```

import lombok.RequiredArgsConstructor;

```

```

import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/clients")
@RequiredArgsConstructor
public class ClientController {

    private final ClientService clientService;

    @GetMapping
    public List<Client> getAllClients() {
        return clientService.getAllClients();
    }

    @GetMapping("/{id}")
    public Client getClientById(@PathVariable Long id) {
        return clientService.getClientById(id);
    }

    @PostMapping
    public Client createClient(@RequestBody Client client) {
        return clientService.saveClient(client);
    }

    @PutMapping("/{id}")
    public Client updateClient(@PathVariable Long id, @RequestBody Client client) {
        return clientService.updateClient(id, client);
    }

    @DeleteMapping("/{id}")
    public void deleteClient(@PathVariable Long id) {
        clientService.deleteClient(id);
    }
}

package ua.opnu.practice1kr.Client;

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

import java.util.Optional;

public interface ClientRepository extends JpaRepository<Client, Long> {
    Optional<Client> findByUsername(String username);
    Optional<Client> findByEmail(String email);
}

```

```

package ua.opnu.practice1kr.Client;

import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Lazy;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import ua.opnu.practice1kr.Client.Role.Role;

import java.util.List;

@Service
@RequiredArgsConstructor
public class ClientService implements UserDetailsService {

    private final ClientRepository clientRepository;
    private final @Lazy PasswordEncoder passwordEncoder;

    @Override
    public UserDetails loadUserByUsername(String username) throws
    UsernameNotFoundException {
        return clientRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException("User not found with
username: " + username));
    }

    public List<Client> getAllClients() {
        return clientRepository.findAll();
    }

    public Client getClientById(Long id) {
        return clientRepository.findById(id).orElseThrow();
    }

    public Client registerNewClient(Client client) {
        if (clientRepository.findByUsername(client.getUsername()).isPresent()) {
            throw new RuntimeException("Username already exists!");
        }
        client.setPassword(passwordEncoder.encode(client.getPassword()));

        if (client.getRole() == null) {
            client.setRole(Role.USER);
        }
        return clientRepository.save(client);
    }

    public Client saveClient(Client client) {
        return clientRepository.save(client);
    }

    public Client updateClient(Long id, Client updatedClient) {

```



```

        Client client = getClientById(id);
        client.setName(updatedClient.getName());
        client.setEmail(updatedClient.getEmail());
        client.setPhone(updatedClient.getPhone());
        return clientRepository.save(client);
    }

    public void deleteClient(Long id) {
        clientRepository.deleteById(id);
    }
}

```

```
package ua.opnu.practice1kr.Client;
```

```

import org.springframework.security.oauth2.client.userinfo.DefaultOAuth2UserService;
import org.springframework.security.oauth2.client.userinfo.OAuth2UserRequest;
import org.springframework.security.oauth2.core.OAuth2AuthenticationException;
import org.springframework.security.oauth2.core.user.OAuth2User;
import org.springframework.stereotype.Service;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.context.annotation.Lazy;
import ua.opnu.practice1kr.Client.Role.Role;

```

```
import java.util.Optional;
```

```
@Service
```

```
public class CustomOAuth2UserService extends DefaultOAuth2UserService {
```

```

    private final ClientRepository clientRepository;
    private final @Lazy PasswordEncoder passwordEncoder;

```

```

    public CustomOAuth2UserService(ClientRepository clientRepository, @Lazy
PasswordEncoder passwordEncoder) {
        this.clientRepository = clientRepository;
        this.passwordEncoder = passwordEncoder;
    }

```

```
@Override
```

```

    public OAuth2User loadUser(OAuth2UserRequest userRequest) throws
OAuth2AuthenticationException {
        OAuth2User oauth2User = super.loadUser(userRequest);

```

```

        String email = oauth2User.getAttribute("email");
        String name = oauth2User.getAttribute("name");

```

```

        Optional<Client> existingClient = clientRepository.findByEmail(email);
        Client client;

```

```

        if (existingClient.isPresent()) {
            client = existingClient.get();
            client.setName(name);

```

```

        client.setAttributes(oauth2User.getAttributes());
        clientRepository.save(client);
    } else {
        client = Client.builder()
            .name(name)
            .email(email)
            .username(email)
            .password(passwordEncoder.encode("GENERATED_OAUTH2_PASSWORD"))
            .role(Role.USER)
            .attributes(oauth2User.getAttributes())
            .build();
        clientRepository.save(client);
    }
    return client;
}
}

```

```

package ua.opnu.practice1kr.Client.Role;

```

```

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

```

```

@RestController
@RequestMapping("/api/admin")
public class AdminController {
    @GetMapping("/dashboard")
    public String adminDashboard() {
        return "Admin dashboard content";
    }
}

```

```

package ua.opnu.practice1kr.Client.Role;

```

```

public enum Role {
    USER,
    ADMIN
}

```

```

package ua.opnu.practice1kr.Client.Role;

```

```

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

```

```

@RestController
@RequestMapping("/api/user")

```

```

public class UserController {
    @GetMapping("/profile")
    public String userProfile() {
        return "User profile content";
    }
}

```

```

package ua.opnu.practice1kr.Config;

```

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Lazy;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import
org.springframework.security.config.annotation.authentication.configuration.AuthenticationCo
nfiguration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import
org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
import ua.opnu.practice1kr.Client.ClientService;
import ua.opnu.practice1kr.Jwt.JwtAuthFilter;
import ua.opnu.practice1kr.Client.CustomOAuth2UserService;

```

```

@Configuration

```

```

@EnableWebSecurity

```

```

public class SecurityConfig {

```

```

    private @Lazy ClientService clientService;
    private final @Lazy JwtAuthFilter jwtAuthFilter;
    private final CustomOAuth2UserService customOAuth2UserService;

```

```

    public SecurityConfig(@Lazy JwtAuthFilter jwtAuthFilter, CustomOAuth2UserService
customOAuth2UserService) {
        this.jwtAuthFilter = jwtAuthFilter;
        this.customOAuth2UserService = customOAuth2UserService;
    }

```

```

    @Autowired

```

```

    public void setClientService(@Lazy ClientService clientService) {
        this.clientService = clientService;
    }

```

@Bean

```
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .csrf(AbstractHttpConfigurer::disable)
        .authorizeHttpRequests(authorize -> authorize
            .requestMatchers("/api/auth/**").permitAll()
            .requestMatchers("/oauth2/**").permitAll()
            .requestMatchers("/api/admin/**").hasAuthority("ADMIN")
            .requestMatchers("/api/user/**").hasAnyAuthority("USER", "ADMIN")
            .requestMatchers("/api/clients").authenticated()
            .requestMatchers("/api/studios").authenticated()
            .requestMatchers("/api/**").authenticated()
            .anyRequest().permitAll()
        )
        .sessionManagement(session -> session
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        )
        .authenticationProvider(authenticationProvider())
        .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class)
        .oauth2Login(oauth2 -> oauth2
            .userInfoEndpoint(userInfo -> userInfo
                .userService(customOAuth2UserService)
            )
            .defaultSuccessUrl("/api/auth/oauth2/success", true)
            .failureUrl("/login?error")
        );

    return http.build();
}
```

@Bean

```
public AuthenticationProvider authenticationProvider() {
    DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
    authProvider.setUserDetailsService(clientService);
    authProvider.setPasswordEncoder(passwordEncoder());
    return authProvider;
}
```

@Bean

```
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

@Bean

```
public AuthenticationManager authenticationManager(AuthenticationConfiguration config)
throws Exception {
    return config.getAuthenticationManager();
}
}
```

```

package ua.opnu.practice1kr.Jwt;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;
import org.springframework.lang.NonNull;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import ua.opnu.practice1kr.Client.ClientService;

import java.io.IOException;

@Component
@RequiredArgsConstructor
public class JwtAuthFilter extends OncePerRequestFilter {

    private final JwtUtil jwtUtil;
    private final ClientService clientService;

    @Override
    protected void doFilterInternal(
        @NonNull HttpServletRequest request,
        @NonNull HttpServletResponse response,
        @NonNull FilterChain filterChain
    ) throws ServletException, IOException {
        final String authHeader = request.getHeader("Authorization");
        final String jwt;
        final String username;

        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return;
        }

        jwt = authHeader.substring(7);
        username = jwtUtil.extractUsername(jwt);

        if (username != null && SecurityContextHolder.getContext().getAuthentication() == null)
        {
            UserDetails userDetails = this.clientService.loadUserByUsername(username);
            if (jwtUtil.isTokenValid(jwt, userDetails)) {
                UsernamePasswordAuthenticationToken authToken = new
                UsernamePasswordAuthenticationToken(
                    userDetails,
                    null,
                    userDetails.getAuthorities()
                );
            }
        }
    }

```

```

        authToken.setDetails(
            new WebAuthenticationDetailsSource().buildDetails(request)
        );
        SecurityContextHolder.getContext().setAuthentication(authToken);
    }
}
filterChain.doFilter(request, response);
}
}

```

```
package ua.opnu.practice1kr.Jwt;
```

```

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.io.Decoders;
import io.jsonwebtoken.security.Keys;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;

```

```

import java.security.Key;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

```

```
@Component
```

```
public class JwtUtil {
```

```

    @Value("${application.security.jwt.secret-key}")
    private String secretKey;
    @Value("${application.security.jwt.expiration}")
    private long jwtExpiration;
    @Value("${application.security.jwt.refresh-token.expiration}")
    private long refreshExpiration;

```

```

    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }

```

```

    public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
    }

```

```

    public String generateToken(UserDetails userDetails) {
        return generateToken(new HashMap<>(), userDetails);
    }

```

```
    public String generateToken(
```

```

        Map<String, Object> extraClaims,
        UserDetails userDetails
    ) {
        return buildToken(extraClaims, userDetails, jwtExpiration);
    }

    public String generateRefreshToken(
        UserDetails userDetails
    ) {
        return buildToken(new HashMap<>(), userDetails, refreshExpiration);
    }

    private String buildToken(
        Map<String, Object> extraClaims,
        UserDetails userDetails,
        long expiration
    ) {
        return Jwts
            .builder()
            .setClaims(extraClaims)
            .setSubject(userDetails.getUsername())
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() + expiration))
            .signWith(getSignInKey(), SignatureAlgorithm.HS256)
            .compact();
    }

    public boolean isTokenValid(String token, UserDetails userDetails) {
        final String username = extractUsername(token);
        return (username.equals(userDetails.getUsername())) && !isTokenExpired(token);
    }

    private boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }

    private Date extractExpiration(String token) {
        return extractClaim(token, Claims::getExpiration);
    }

    private Claims extractAllClaims(String token) {
        return Jwts
            .parserBuilder()
            .setSigningKey(getSignInKey())
            .build()
            .parseClaimsJws(token)
            .getBody();
    }

    private Key getSignInKey() {
        byte[] keyBytes = Decoders.BASE64.decode(secretKey);
        return Keys.hmacShaKeyFor(keyBytes);
    }

```

```
    }  
}
```

```
import jakarta.persistence.*;
```

```
import lombok.*;
```

```
@Entity
```

```
@Data
```

```
@NoArgsConstructor
```

```
@AllArgsConstructor
```

```
@Builder
```

```
public class Mechanic {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    private String name;
```

```
    private String specialization;
```

```
}
```

```
package ua.opnu.practice1kr.Mechanic;
```

```
import lombok.RequiredArgsConstructor;
```

```
import org.springframework.web.bind.annotation.*;
```

```
import java.util.List;
```

```
@RestController
```

```
@RequestMapping("/api/mechanics")
```

```
@RequiredArgsConstructor
```



```

public class MechanicController {

    private final MechanicService service;

    @GetMapping
    public List<Mechanic> getAll() {
        return service.getAll();
    }

    @GetMapping("/{id}")
    public Mechanic get(@PathVariable Long id) {
        return service.getById(id);
    }

    @PostMapping
    public Mechanic create(@RequestBody Mechanic m) {
        return service.save(m);
    }

    @PutMapping("/{id}")
    public Mechanic update(@PathVariable Long id, @RequestBody Mechanic m) {
        return service.update(id, m);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        service.delete(id);
    }
}

```

```

package ua.opnu.practice1kr.Mechanic;

```

```

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

```

```

public interface MechanicRepository extends JpaRepository<Mechanic, Long> {
}

```

```

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

```

```

import java.util.List;

```

```

@Service
@RequiredArgsConstructor
public class MechanicService {

```

```

    private final MechanicRepository mechanicRepository;

```

```

    public List<Mechanic> getAll() {

```

```

        return mechanicRepository.findAll();
    }

    public Mechanic getById(Long id) {
        return mechanicRepository.findById(id).orElseThrow();
    }

    public Mechanic save(Mechanic mechanic) {
        return mechanicRepository.save(mechanic);
    }

    public Mechanic update(Long id, Mechanic updated) {
        Mechanic mechanic = getById(id);
        mechanic.setName(updated.getName());
        mechanic.setSpecialization(updated.getSpecialization());
        return mechanicRepository.save(mechanic);
    }

    public void delete(Long id) {
        mechanicRepository.deleteById(id);
    }
}

```

```
package ua.opnu.practice1kr.ServiceRecord;
```

```
import jakarta.persistence.*;
import lombok.*;
import ua.opnu.practice1kr.Car.Car;
import ua.opnu.practice1kr.Mechanic.Mechanic;
import ua.opnu.practice1kr.ServiceType.ServiceType;
```

```
import java.time.LocalDate;
```

```
@Entity
```

```
@Data
```

```
@NoArgsConstructor
```

```
@AllArgsConstructor
```

```
@Builder
```

```
public class ServiceRecord {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    @ManyToOne
```

```
    @JoinColumn(name = "car_id")
```

```
    private Car car;
```

```
    @ManyToOne
```

```
    @JoinColumn(name = "mechanic_id")
```

```
    private Mechanic mechanic;
```

```

@ManyToOne
@JoinColumn(name = "service_type_id")
private ServiceType serviceType;

private LocalDate date;

private String description;
}

```

```

import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.*;
import ua.opnu.practice1kr.ServiceType.ServiceType;
import ua.opnu.practice1kr.ServiceType.ServiceTypeService;

```

```

import java.math.BigDecimal;
import java.time.LocalDate;
import java.util.List;

```

```

@RestController
@RequestMapping("/api/records")
@RequiredArgsConstructor
public class ServiceRecordController {

```

```

    private final ServiceRecordService service;
    private final ServiceTypeService serviceTypeService;

```

```

    @GetMapping
    public List<ServiceRecord> getAll() {
        return service.getAll();
    }

```

```

    @GetMapping("/{id}")
    public ServiceRecord get(@PathVariable Long id) {
        return service.getById(id);
    }

```

```

    @GetMapping("/car/{carId}")
    public List<ServiceRecord> byCar(@PathVariable Long carId) {
        return service.getByCarId(carId);
    }

```

```

    @GetMapping("/mechanic/{mechanicId}")
    public List<ServiceRecord> byMechanic(@PathVariable Long mechanicId) {
        return service.getByMechanicId(mechanicId);
    }

```

```

    @PostMapping
    public ServiceRecord create(@RequestBody ServiceRecord r) {
        return service.save(r);
    }

```

```

    }

    @PutMapping("/{id}")
    public ServiceRecord update(@PathVariable Long id, @RequestBody ServiceRecord r) {
        return service.update(id, r);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        service.delete(id);
    }

    @PutMapping("/{recordId}/assign-service-type/{serviceTypeId}")
    public ServiceRecord assignServiceType(@PathVariable Long recordId, @PathVariable
Long serviceTypeId) {
        ServiceType serviceType = serviceTypeService.getById(serviceTypeId);
        return service.assignServiceType(recordId, serviceType);
    }

    @GetMapping("/car/{carId}/total-cost")
    public BigDecimal getTotalCostByCarId(@PathVariable Long carId) {
        return service.getTotalCostByCarId(carId);
    }

    @GetMapping("/mechanic/{mechanicId}/count")
    public long countRecordsByMechanic(@PathVariable Long mechanicId) {
        return service.countRecordsByMechanicId(mechanicId);
    }

    @GetMapping("/popular-service-types")
    public List<Object[]> getMostPopularServiceTypes() {
        return service.getMostPopularServiceTypes();
    }

    @GetMapping("/between")
    public List<ServiceRecord> getRecordsBetween(
        @RequestParam("start") LocalDate start,
        @RequestParam("end") LocalDate end) {
        return service.getRecordsBetweenDates(start, end);
    }

    @GetMapping("/top-cars")
    public List<Object[]> getTopServicedCars() {
        return service.getTopServicedCars();
    }
}

```

```

package ua.opnu.practice1kr.ServiceRecord;

```

```

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

```

```

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

import java.time.LocalDate;
import java.util.List;

public interface ServiceRecordRepository extends JpaRepository<ServiceRecord, Long> {
    List<ServiceRecord> findByCarId(Long carId);
    List<ServiceRecord> findByMechanicId(Long mechanicId);

    @Query("SELECT sr.serviceType.name, COUNT(sr) as cnt FROM ServiceRecord sr GROUP BY sr.serviceType.name ORDER BY cnt DESC")
    List<Object[]> countServiceTypeUsage();

    List<ServiceRecord> findByDateBetween(LocalDate startDate, LocalDate endDate);

    @Query("SELECT sr.car.id, COUNT(sr) as cnt FROM ServiceRecord sr GROUP BY sr.car.id ORDER BY cnt DESC")
    List<Object[]> countServiceRecordsByCar();
}

```

```

package ua.opnu.practice1kr.ServiceRecord;

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import ua.opnu.practice1kr.ServiceType.ServiceType;

import java.math.BigDecimal;
import java.time.LocalDate;
import java.util.List;

@Service
@RequiredArgsConstructor
public class ServiceRecordService {

    private final ServiceRecordRepository serviceRecordRepository;

    public List<ServiceRecord> getAll() {
        return serviceRecordRepository.findAll();
    }

    public ServiceRecord getById(Long id) {
        return serviceRecordRepository.findById(id).orElseThrow();
    }

    public List<ServiceRecord> getByCarId(Long carId) {
        return serviceRecordRepository.findByCarId(carId);
    }

    public List<ServiceRecord> getByMechanicId(Long mechanicId) {
        return serviceRecordRepository.findByMechanicId(mechanicId);
    }
}

```

```

    }

    public ServiceRecord save(ServiceRecord record) {
        return serviceRecordRepository.save(record);
    }

    public ServiceRecord update(Long id, ServiceRecord updated) {
        ServiceRecord record = getById(id);
        record.setDate(updated.getDate());
        record.setDescription(updated.getDescription());
        record.setCar(updated.getCar());
        record.setMechanic(updated.getMechanic());
        return serviceRecordRepository.save(record);
    }

    public void delete(Long id) {
        serviceRecordRepository.deleteById(id);
    }

    public ServiceRecord assignServiceType(Long recordId, ServiceType serviceType) {
        ServiceRecord record = getById(recordId);
        record.setServiceType(serviceType);
        return serviceRecordRepository.save(record);
    }

    public BigDecimal getTotalCostByCarId(Long carId) {
        List<ServiceRecord> records = serviceRecordRepository.findByCarId(carId);
        return records.stream()
            .filter(r -> r.getServiceType() != null)
            .map(r -> r.getServiceType().getStandardPrice())
            .reduce(BigDecimal.ZERO, BigDecimal::add);
    }

    public long countRecordsByMechanicId(Long mechanicId) {
        return serviceRecordRepository.findByMechanicId(mechanicId).size();
    }

    public List<Object[]> getMostPopularServiceTypes() {
        return serviceRecordRepository.countServiceTypeUsage();
    }

    public List<ServiceRecord> getRecordsBetweenDates(LocalDate start, LocalDate end) {
        return serviceRecordRepository.findByDateBetween(start, end);
    }

    public List<Object[]> getTopServicedCars() {
        return serviceRecordRepository.countServiceRecordsByCar();
    }
}

```

```

package ua.opnu.practice1kr.ServiceType;

import jakarta.persistence.*;
import lombok.*;

import java.math.BigDecimal;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class ServiceType {

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

    private String name;

    private BigDecimal standardPrice;
}

```

```

package ua.opnu.practice1kr.ServiceType;

import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/types")
@RequiredArgsConstructor
public class ServiceTypeController {

    private final ServiceTypeService service;

    @GetMapping
    public List<ServiceType> getAll() {
        return service.getAll();
    }

    @GetMapping("/{id}")
    public ServiceType get(@PathVariable Long id) {
        return service.getById(id);
    }

    @PostMapping
    public ServiceType create(@RequestBody ServiceType type) {
        return service.save(type);
    }
}

```

```

    @PutMapping("/{id}")
    public ServiceType update(@PathVariable Long id, @RequestBody ServiceType type) {
        return service.update(id, type);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        service.delete(id);
    }
}

```

```

package ua.opnu.practice1kr.ServiceType;

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

public interface ServiceTypeRepository extends JpaRepository<ServiceType, Long> {
}

```

```

package ua.opnu.practice1kr.ServiceType;

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
@RequiredArgsConstructor
public class ServiceTypeService {

    private final ServiceTypeRepository repository;

    public List<ServiceType> getAll() {
        return repository.findAll();
    }

    public ServiceType getById(Long id) {
        return repository.findById(id).orElseThrow();
    }

    public ServiceType save(ServiceType serviceType) {
        return repository.save(serviceType);
    }

    public ServiceType update(Long id, ServiceType updated) {
        ServiceType s = getById(id);
        s.setName(updated.getName());
        s.setStandardPrice(updated.getStandardPrice());
        return repository.save(s);
    }
}

```



```
    }  
  
    public void delete(Long id) {  
        repository.deleteById(id);  
    }  
}
```