ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ

ФАКУЛТЕТ КОМПЮТЪРНИ СИСТЕМИ И ТЕХНОЛОГИИ

КУРСОВА РАБОТА

Дисциплина: „Съвременни Java технологии”

тема: „Разработване на сървърно приложение, което да поддържа база данни от произведените автомобили в автомобилна фабрика“

***Изготвил**:*
Иван Стоянов Панайотов
Фак. № 121325022
Група: 221
КСИ
e-mail: ipajotov@tu-sofia.bg

София, 2025

Съдържание

1. Тема

CarFactory е сървърно приложение, реализирано като уеб услуга, предназначено да съхранява произведените автомобили от автомобилна фабрика. Автомобилната фабрика може да произвежда всякакви автомобили от какъвто и да е бранд и модел. Всеки един автомобил трябва да има свои уникален идентификационен номер, както и регистрационен номер, който да спазва българският стандарт за регистрационен номер. Освен тези характеристики, всеки един автомобил има и цена. Чрез приложението се осигурява централизирано съхранение, обработка и достъп до данни за произведените автомобили. Цялата информация ще се съхранява в база данни PostgreSQL. За комуникация между базата данни и back-end приложението ще се използват ORM връзки. Комуникацията между back-end приложението и клиента ще се осъществява чрез CRUD операции. За достъп до цялата система, автомобилният производител ще има нужда да се запознае с предварително дефинираните данни за вход в системата (JWT-базирана автентикация).

2. Цели и задачи

Основните цели в проекта са:

- Изграждане на Api за управление на произведените автомобили.
- Реализация на пълни CRUD операции.
- Използване на ORM за работа с релационна база данни.
- Осигуряване на валидност и цялостност на входящите данни.
- Реализиране на сигурност чрез JWT авторизация.

Основните задачи в проекта са:

- Дефиниране на модел (Entity) на автомобил.
- Свързване с PosgreSQL база данни.
- Осигуряване на достъп чрез REST контролери.
- Централизирана обработка на грешки при въвеждане на данни от клиента.
- Ограничаване на достъпа чрез токен-базирана автентикация.

3. Функционалности

Проектът CarFactory има възможностите изброени по-долу:

- Добавяне на автомобил с валидация на входните данни. Бранда и модела на автомобила трябва да бъдат минимум 2 знака, като специални символи не са позволени. Регистрационният номер на автомобила трябва да спазва българския формат за регистрационен номер. Цената която се въвежда трябва да бъде по-голяма от 0 лева.
- Изваждане на списък с произведени автомобили по дадена марка.
- Редактиране на което и да е поле от даден автомобил. Като параметър се подава уникалният идентификационен номер.
- Изтриване на автомобил по неговият идентификационен номер в базата данни.

- Валидация на входните данни.
- JWT автентикация.
- Разбираеми съобщения при грешка при верификация.

4. Използвани технологии

Категория	Технология
Програмен език	Java
Framework	Spring Boot
ORM	Spring Data JPA
База данни	PostgreSQL
Валидация	Jakarta Validation
REST	Spring Web
Сигурност	JWT
Тестова среда	Postman
Среда за разработка	VS Code

5. Архитектура на проекта

Проектът следва многослойна архитектура както е показано на Фиг. 5.1. Това позволява ясно разделение между различните отговорности в системата, по-лесна поддръжка и гъвкавост при надграждане. Основната логика е структурирана в четири слоя: Controller, Service, Repository и Domain слой. Всеки от тях изпълнява конкретна роля и взаимодейства с останалите по строго дефиниран начин.

5.1. Слой Контролер

Контролера са входната точка за всички HTTP заявки. Те приемат информацията от клиента (например JSON структура), валидират входните данни чрез анотации като @Valid и предават заявката към Service слоя. Контролерът не съдържа бизнес логика, а само координира комуникацията, Фиг 5.2.

```
24    @GetMapping("/brand")
25    public ResponseEntity<?> byBrand(@RequestParam String brand) {
26
27        List<Car> cars = service.findByBrand(brand);
28
29        if (cars.isEmpty()) {
30            return ResponseEntity.status(status: 404)
31                    .body(Map.of("error", "No cars found for brand: " + brand));
32        }
33
34        return ResponseEntity.ok(cars);
35    }
```

Фиг. 5.2. Част от програмният код на контролер слоят

Контролерът обработва само HTTP аспектите – маршрути, параметри, статус кодове и структурата на отговора.

5.2. Service слой

На Фиг. 5.3. е показан част от кода на Service слоят. Тук се намира цялата бизнес логика — валидиране, обработка на данните, трансформации, проверки и специални правила. Слоят гарантира, че логиката е независима от начина на представяне на данните от графичният дизайн и начина на съхранението им в базата данни.

```
31      public Optional<Car> update(long id, Car patch) {
32          return repo.findById(id).map(existing -> {
33            if (patch.getBrand() != null) existing.setBrand(patch.getBrand());
34              if (patch.getModel() != null) existing.setModel(patch.getModel());
35              if (patch.getRegistrationNumber() != null) existing.setRegistrationNumber(patch.getRegistrationNumber());
36              if (patch.getPrice() != null) existing.setPrice(patch.getPrice());
37              return repo.save(existing);
38          });
39      }
```

Фиг 5.3. Част от кода на Service слоя

5.3. Repository слой

Repository слоят отговаря за взаимодействието с базата данни. Той използва Spring Data JPA, което позволява автоматична генерация на SQL заявки чрез имена на методи, както е показано на фиг 5.4. Също по този начин базата данни е изолирана от останалата част от приложението.

```
8   @Repository
9   public interface CarRepository extends JpaRepository<Car, Long> {
10      List<Car> findByBrandIgnoreCase(String brand);
11  }
```

Фиг. 5.4. Част от кода на Repository слоя

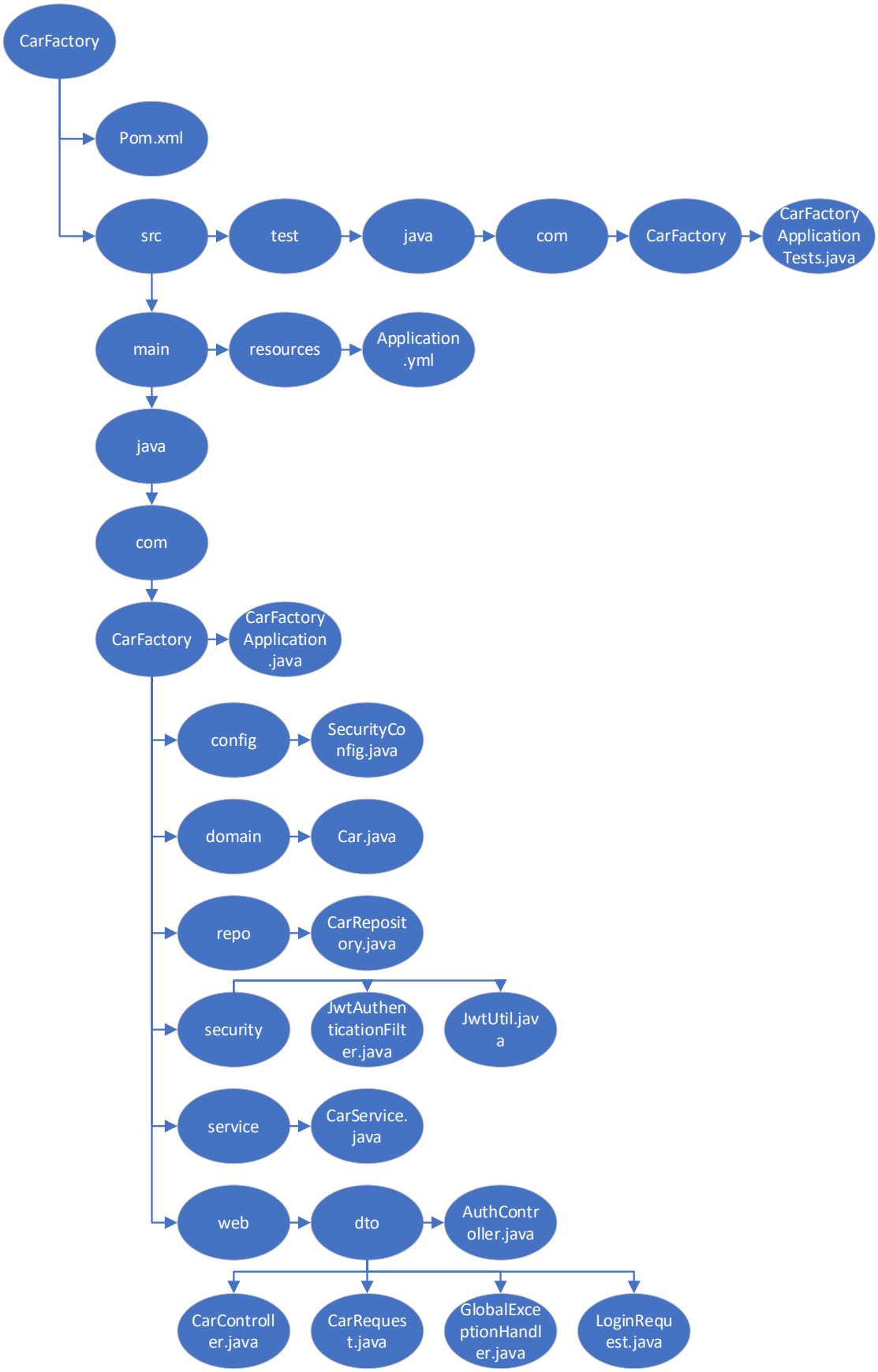
По този начин Spring Boot автоматично създава SQL заявката:
*SELECT * FROM car WHERE LOWER(brand) = LOWER(?);*

5.4. Domain слой

Този слой съдържа обектите, които се картографират към таблици в базата. Тук е реализиран ORM моделът — всяко поле от класа съответства на колона в таблицата. Тези класове представляват основните бизнес обекти, с които работи приложението. Всеки обект Car автоматично се превръща в запис в таблица car.

```
14      @Id
15      @GeneratedValue(strategy = GenerationType.IDENTITY)
16      private Long id;
17
18      @Column(nullable = false)
19      @Size(min = 2, message = "Brand must be at least 2 characters")
20      private String brand;
21
22      @Column(nullable = false)
23      @Size(min = 2, message = "Model must be at least 2 characters")
24      private String model;
25
26      @Column(unique = true, nullable = false)
27      @Pattern(regexp = "^[A-Z]{2} \\d{4} [A-Z]{2}$",message = "Registration number must follow the format 'AA 1234 BB'")
28      private String registrationNumber;
```

Фиг. 5.5. Част от кода на Domain слоя



Фиг. 5.1. Архитектура на CarFactory

5.5. Обработка и валидация на грешки

Приложението използва специален слой за централизирано прихващане на грешки чрез @ControllerAdvice. Този метод позволява да се изпращат унифицирани съобщения към потребителя, адекватни отговори при грешни входни грешни входни данни, както и прихващане на SQL грешки. Примерен код за това е даден на фиг. 5.6.

```
@ExceptionHandler(DataIntegrityViolationException.class)
public ResponseEntity<Object> handleDataIntegrityViolationException(DataIntegrityViolationException ex) {
    String message = ex.getMostSpecificCause().getMessage();

    if (message != null && message.contains("registration_number")) {
        message = "Registration number is already in use";
    } else {
        message = "Database error: " + message;
    }

    Map<String, String> response = new HashMap<>();
    response.put("error", message);

    return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
}
```

Фиг. 5.6. Обработка и валидация на грешки

5.6. JWT Security слой

Като допълнение в проекта е добавен и слой за сигурност, чрез Spring Secirity и JWT. Този слой издава токен, чрез който прозводителят да може да обработва заявки, филтрира като проверява токена преди достъп до защитените ресурси. Защитените ресурси се определят от класът SecurityConfig, който дефинира точно тези маршрути. Това изгражда надеждна и стандартна авторизационна архитектура.

```
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

    http
        .csrf(csrf -> csrf.disable())
        .authorizeHttpRequests(auth -> auth
            .requestMatchers(...patterns: "/auth/**").permitAll()
            .requestMatchers(...patterns: "/api/cars/**").authenticated()
            .anyRequest().permitAll()
        )
        .addFilterBefore(jwtFilter, beforeFilter: UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
}
```

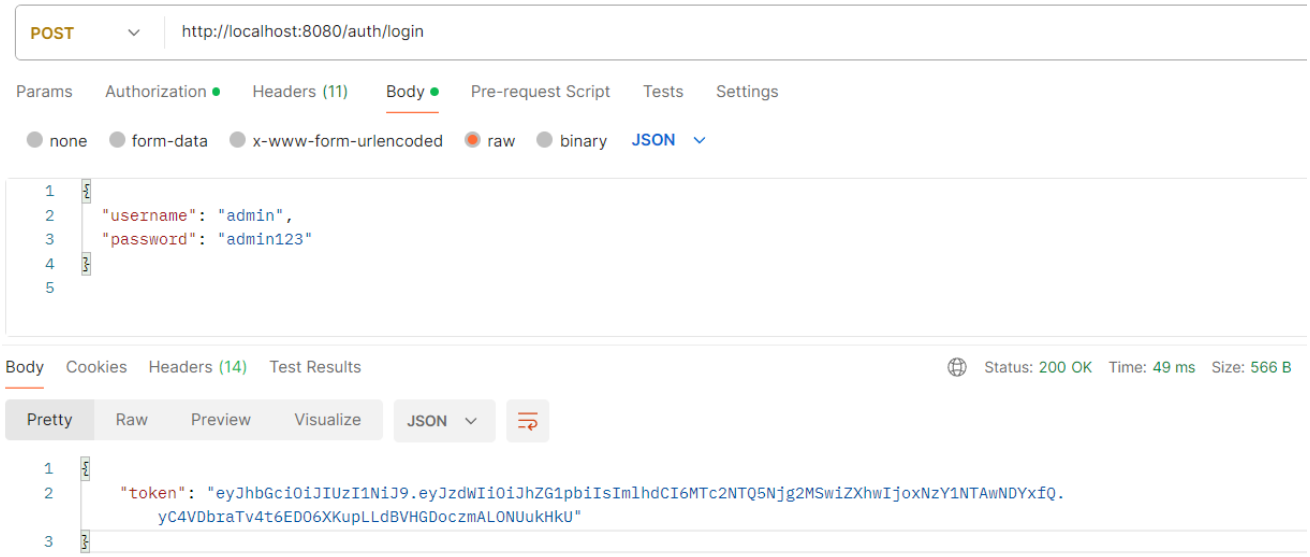
Фиг. 5.7. Част от кода на слоят за сигурност

6. Примери за работа с приложението

6.1. Автентикиране на автомобилен производител в приложението:

Работата с приложението започва с извършването на процес по удостоверяване, тъй като всички последващи заявки изискват валиден JWT токен. Потребителят първо изпраща заявка за вход с POST заявка на адрес

http://localhost:8080/auth/login, като предоставя потребителско име и парола във формат JSON. След успешна автентикация системата генерира JWT токен, който се връща като част от отговора. Този токен трябва да бъде копиран и поставен в заглавието *Authorization* на всички следващи заявки, като се използва формат *Bearer {token}*. На Фиг. 6.1 е показан примерен отговор от успешен login, съдържащ генерирания токен.



Фиг. 6.1. Request и Response на POST заявка за авторизация чрез Postman

Входни данни за стъпката:
POST http://localhost:8080/auth/login *Content-Type: application/json* *{* *"username": "admin",* *"password": "admin123"* *}*

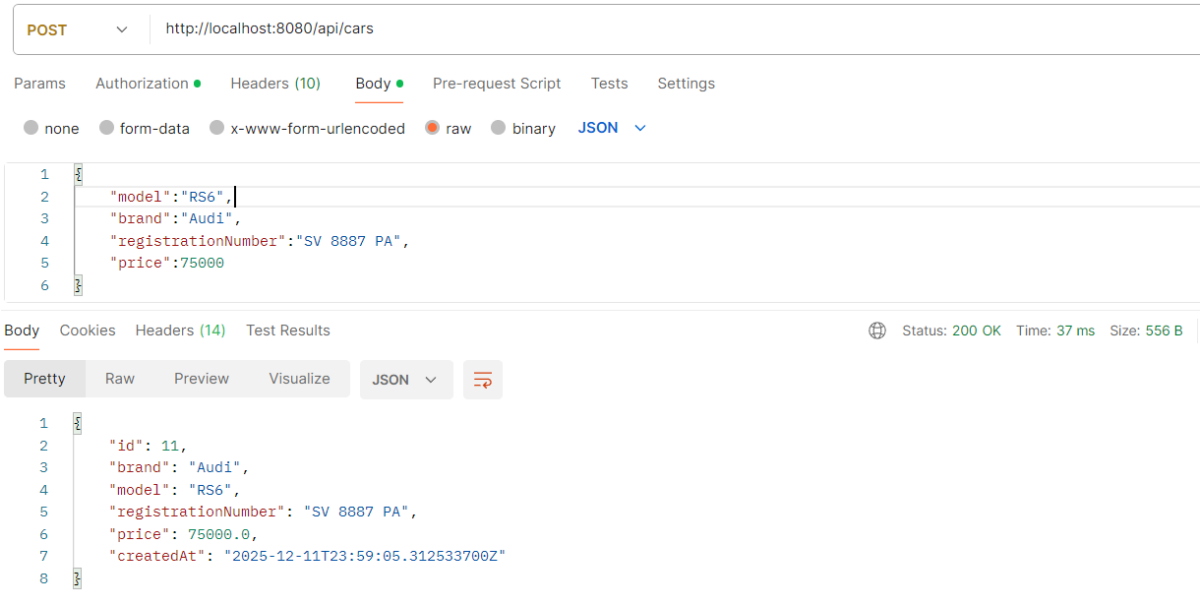
След като токенът е наличен, потребителят може да извършва CRUD операции върху ресурсите в системата. Всички заявки към приложението работят изцяло в JSON формат.

6.2. Добавяне на автомобил

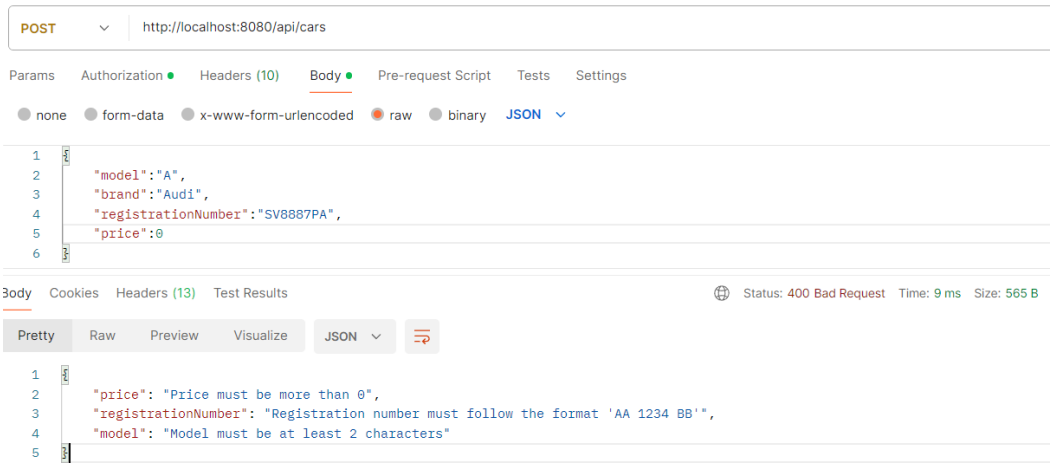
На Фиг. 6.2. и Фиг 6.3 е показан съответно пример за коректно и некоректно подадено JSON тяло при създаване на автомобил.

Входните данни са както следва:
POST http://localhost:8080/api/cars *Content-Type: application/json* *Authorization: Bearer {token}*

```
{
    "model":"RS6",
    "brand":"Audi",
    "registrationNumber":"SV 8887 PA",
"price":75000
}
```



Фиг. 6.2. Request и Response на POST заявка за добавяне на нов автомобил чрез Postman

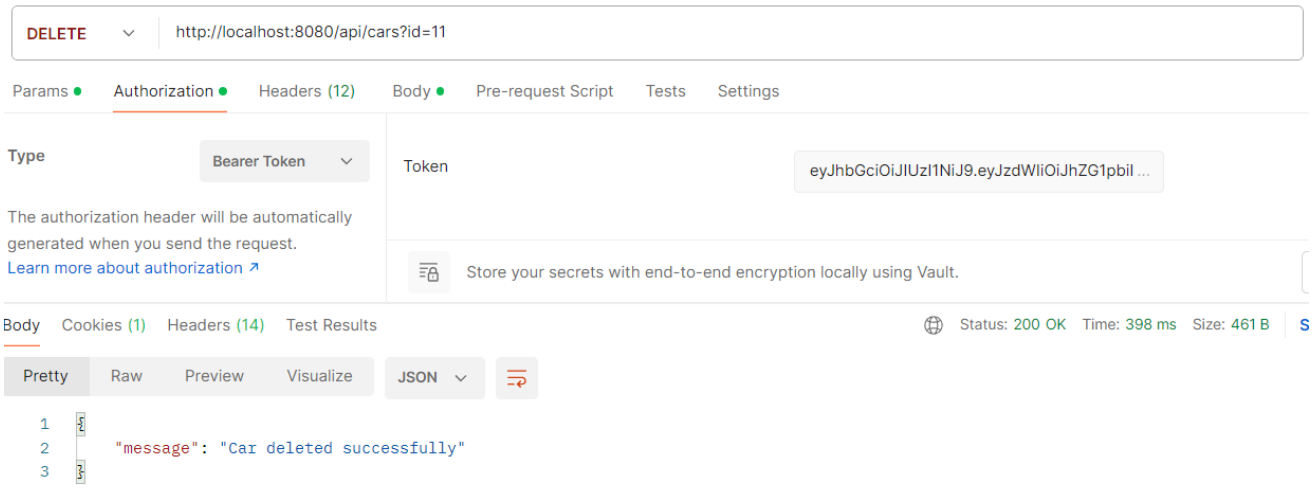


Фиг. 6.3. Неуспешен опит за регистриране на автомобил

6.3. Изтриване на автомобил

Изтриването на автомобил става чрез *DELETE* заявка към *http://localhost:8080/api/cars?id={id}*. Системата проверява дали автомобил със съответния идентификатор съществува и връща съобщение дали операцията е успешна. При успешното изтриване системата връща потвърждение, а при неуспех – съобщение, че автомобилът не е намерен. Пример за това е показан на Фигура 6.4.

Входните данни са както следва:
POST http://localhost:8080/api/cars?id=11 *Content-Type: application/json* *Authorization: Bearer {token}*

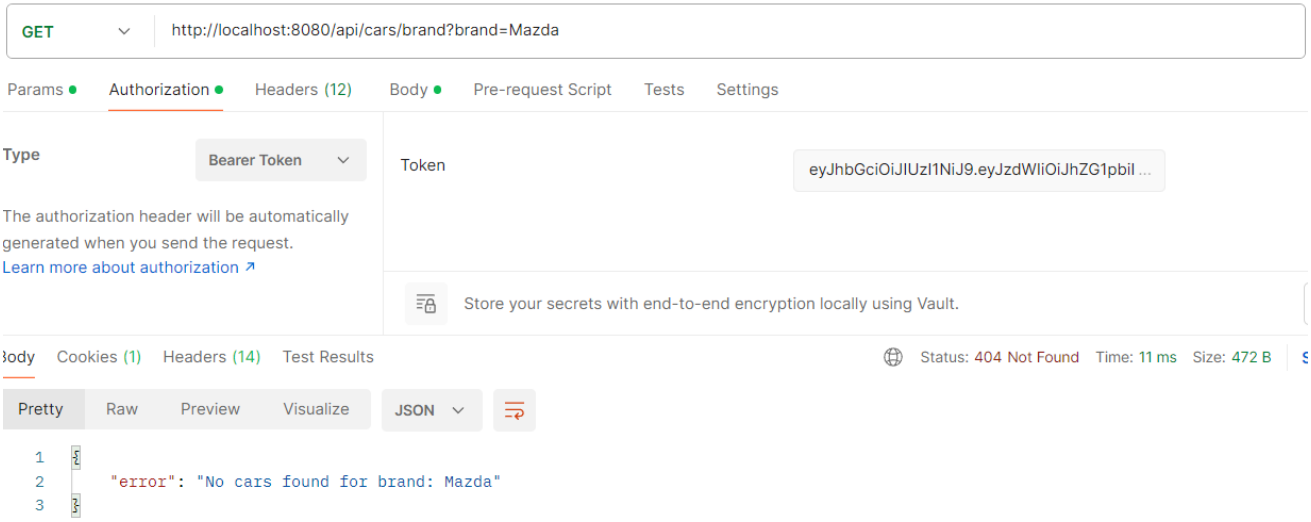


Фиг. 6.4. Request и Response на DELETE заявка за изтриване на автомобил чрез Postman

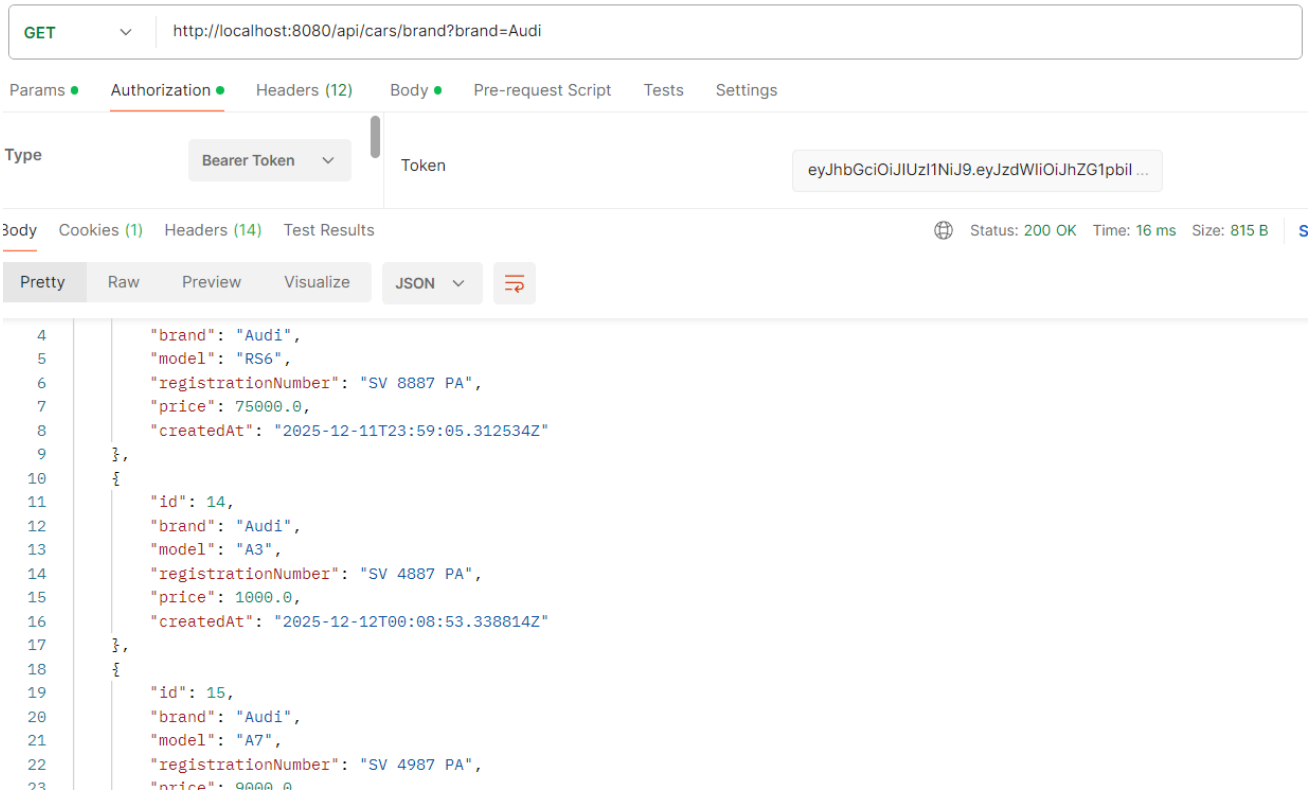
6.4. Търсене по марка

За извличане на информация за произведените автомобили от конкретен бранд може да бъде използвана *GET* заявка към *http://localhost:8080/api/cars/brand?brand={BrandName}*. При наличие на валиден токен системата връща списък с всички автомобили от този производител в JSON формат. Ако списъкът е празен, приложението връща статус код 404, както и отговор чрез Json. Примерни резултати от заявката са показани на Фиг. 6.5 и Фиг.6.6.

Входните данни са както следва:
POST http://localhost:8080/api/cars/brand?brand=Audi *Content-Type: application/json* *Authorization: Bearer {token}*



Фиг. 6.4. Request и Response на GET заявка за извличане на автомобили чрез Postman

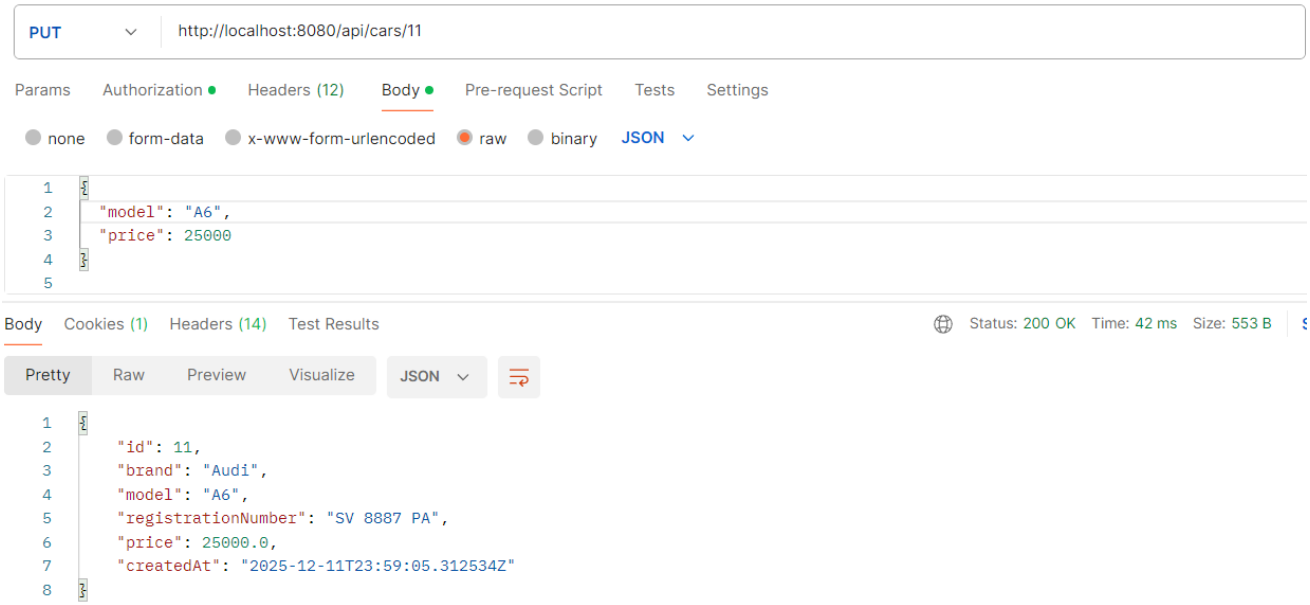


Фиг. 6.5. Request и Response на GET заявка за извличане на автомобили чрез Postman

6.5. Актуализиране на данни за вече въведен автомобил

Редакцията на съществуващ автомобил се извършва чрез *PUT* заявка към http://localhost:8080/api/*cars/{id}*, където се подава идентификаторът на автомобила. В тялото на заявката отново се изпраща JSON обект със стойностите, които трябва да бъдат актуализирани. Ако автомобилът съществува, системата връща обновения запис, иначе се връща JSON съобщение за грешка. На Фиг. 6.6 е даден пример за изпратена заявка за редакция.

Входните данни са както следва:
POST http://localhost:8080/api/cars/11 *Content-Type: application/json* *Authorization: Bearer {token}* *{* *"model": "A6",* *"price": 25000* *}*



Фиг. 6.6. Request и Response на PUT заявка за актуализиране на автомобил чрез Postman