Car Dealership – User Manual

🎓 **South East European University – Skopje**
CST Faculty - Programming in Java

🏫 Lecturer: **Nuhi Besimi**

👩‍💻 Members:
Argjend Mehmedi (am28507@seeu.edu.mk)
Kan Misini (km28487@seeu.edu.mk)

Introduction

The Car Dealership application allows you to store and manage data relevant to a car dealership. It's a simple project to start understanding the logic behind creating and maintaining a system for a dealership, and also understanding the technologies used in this project.

This documentation describes how the application is implemented and its principles, how you can run it, the dependencies used, the REST endpoints, and more.

Getting Started

To start, download the branch provided in GitHub and preferably use IntelliJ to open the project. The project is a Spring Boot application using Maven with Java 11 SDK. All of the modules needed should automatically download when you first open the project.

Activate XAMPP and run the MySQL and Apache Servers. After you run the Spring Boot application, you can access the application at `http://localhost:8000/` through Postman.

Structure

Here's a detailed view of the project's whole structure:

Database

Our database consists of 5 entities:

Car

VIN – PK	brand	model	year

Customer

customerid – PK	firstname	lastname	address	phonenr

Dealership

dealershipid – PK	city	address

Employee

employeeid – PK	firstname	lastname

Sale

invoicenr – PK	vin – FK	customerid – FK

MySQL DDL Scripts

```
CREATE TABLE `Car` (
  `vin` int(30) NOT NULL,
  `brand` varchar(40) NOT NULL,
  `model` varchar(40) NOT NULL,
  `year` int(10) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

ALTER TABLE `Car`
  ADD PRIMARY KEY (`vin`);
```

```
CREATE TABLE `Customer` (
  `customerid` int(10) NOT NULL,
  `firstname` varchar(40) NOT NULL,
  `lastname` varchar(40) NOT NULL,
  `address` varchar(40) NOT NULL,
  `phonenr` int(20) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

ALTER TABLE `Customer`
  ADD PRIMARY KEY (`customerid`);
```

```
CREATE TABLE `Dealership` (
  `dealershipid` int(10) NOT NULL,
  `city` varchar(50) NOT NULL,
  `address` varchar(50) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

ALTER TABLE `Dealership`
  ADD PRIMARY KEY (`dealershipid`);
```

```
CREATE TABLE `Employee` (
  `employeeid` int(10) NOT NULL,
  `firstname` varchar(40) NOT NULL,
  `lastname` varchar(40) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

ALTER TABLE `Employee`
  ADD PRIMARY KEY (`employeeid`);
```

```
CREATE TABLE `Sale` (
  `customerid` int(10) NOT NULL,
  `vin` int(30) NOT NULL,
  `invoicenr` int(15) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

ALTER TABLE `Sale`
  ADD PRIMARY KEY (`invoicenr`),
  ADD KEY `sale_ibfk_2` (`customerid`),
  ADD KEY `sale_ibfk_3` (`vin`);

ALTER TABLE `Sale`
  ADD CONSTRAINT `sale_ibfk_2` FOREIGN KEY (`customerid`) REFERENCES `Customer` (`custom
erid`) ON DELETE CASCADE ON UPDATE CASCADE,
  ADD CONSTRAINT `sale_ibfk_3` FOREIGN KEY (`vin`) REFERENCES `Car` (`VIN`) ON DELETE CA
SCADE ON UPDATE CASCADE;
COMMIT;
```

Dependencies & pom.xml

We've used the basic dependencies needed to run this application:

Spring Boot Data JPA

Persists data in SQL stores with Java Persistence API using Spring Data and Hibernate.

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

Spring Web

Used to build web, including RESTful, applications using Spring MVC.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

MySQL Driver

Driver for MySQL JDBC and R2DBC - used to connect the application with the database.

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

Application Properties

The application runs on the `port 8000`. This can be changed to whatever port you aren't using. Also, the database has a password on it. We didn't create an environment variable since we have no plans to publish this project to the public. If we were to publish it, creating an environment variable is crucial to the security of the database and application in general.

```
server.port = 8000

spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/dealership
spring.datasource.username=root
spring.datasource.password=12345678

spring.jpa.hibernate.ddl-auto=none
spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNa
mingStrategyStandardImpl
```

Packages

The project is structured in multiple packages in order to minimize confusion and complexity:

- **controller**
  - controllers
- **pojo**
  - **entity**

        - entity classes
    - **input**
        - input classes
- **repository**
    - repo interfaces
- **services**
    - **defaultservices**
        - defaultservices classes
    - services interfaces

Entities

For every table in our database, we have created a corresponding entity class. In each entity class, we have the table attributes as variables, constructors, getter and setter methods for each variable. For example:

```
@Entity
@Table(name = "Car")
public class Car {
    @Id
    @Column(name = "vin", nullable = false)
    private Integer vin;

    @Column(name = "brand")
    private String brand;

    @Column(name = "model")
    private String model;

    @Column(name = "year")
    private Integer year;

    @OneToOne(mappedBy = "car")
    @JoinColumn(name = "invoicenr")
    @JsonBackReference
    private Sale sale;

    // Constructors
    public Car() {
        this.vin = 0;
        this.brand = "N/A";
        this.model = "N/A";
        this.year = 0;
    }
    public Car(int vin, String brand, String model, int year) {
```

```
        this.vin = vin;
        this.brand = brand;
        this.model = model;
        this.year = year;
    }

    // Getters and setters
    public Integer getVin() { return vin; }
    public void setVin(Integer vin) { this.vin = vin; }

    public String getBrand() { return brand; }
    public void setBrand(String brand) { this.brand = brand; }

    public String getModel() { return model; }
    public void setModel(String model) { this.model = model; }

    public Integer getYear() { return year; }
    public void setYear(Integer year) { this.year = year; }
}
```

To let Spring know which variable is the primary key, we have used the `@Id` annotation tag above the variable.

In the Car Entity class, you can notice that we have used the Car class as a foreign key in the Sale class. We have a One-To-One relationship in this case, and we use the `@JsonBackReference` annotation to indicate that this property is part of a two-way linkage between fields and that property is the backlink:

```
@OneToOne(mappedBy = "car")
@JoinColumn(name = "invoicenr")
@JsonBackReference
private Sale sale;
```

Additionally, we have to implement the other part of the linkage in the Sale class:

```
@OneToOne
@JoinColumn(name = "customerid")
private Customer customer;

@OneToOne
@JoinColumn(name = "vin")
private Car car;
```

We have two foreign key constraints in the Sale class, one being a `Customer object (customerid)` and the other a `Car object (vin)`, with One-To-One relationships.

Services

We use Services in Spring Boot to write the layer with the business logic. It’s used as a middle layer to link the controller with the repository. Service interfaces are firstly used to define the methods we are going to use, and then we implement them in the DefaultServices classes where we store, retrieve, update, and delete data.

```
public interface CarService {
    List<Car> getAllCars();
    Car findCarByVin(Integer vin);
    List<Car> findCarsByBrand(String brand);
    List<Car> findCarsByModel(String model);
    List<Car> findCarsByYear(Integer year);
    Car insertCar(CarInput carInput);
    Car updateCar(CarInput carInput, Integer vin);
    Boolean deleteCar(Integer vin);
}
```

```
@Override
public Car insertCar(CarInput carInput) {
    Car car = new Car(carInput.getVin(), carInput.getBrand(), carInput.getModel(), carIn
put.getYear());
    return carRepository.save(car);
}
```

```
@Override
public Car updateCar(CarInput carInput, Integer vin) {
    Car car = carRepository.findCarByVin(vin);
    if (car == null) { return null; }
    car.setVin(carInput.getVin());
    car.setBrand(carInput.getBrand());
    car.setModel(carInput.getModel());
    car.setYear(carInput.getYear());
    return carRepository.save(car);
}
```

```
@Override
public Boolean deleteCar(Integer vin) {
    Car car = carRepository.findCarByVin(vin);
    if (car != null) { carRepository.delete(car); }
    return true;
}
```

The services for the remaining entities are implemented in a similar manner.

Controllers

In the controllers, you have the different REST requests and required mappings. Every entity has at least a single GET, POST, PUT and DELETE mapping. For example:

```
// GET Request - Receive all car instances
@GetMapping("/cars")
public List<Car> getAllCars() {
    return carService.getAllCars();
}

/**
* GET Request - Receive the car with a specific VIN
* @param vin - VIN of the car
* @return - The car with that VIN
*/
@GetMapping("/cars/{vin}")
public Car findCarByVin(@PathVariable Integer vin) {
    return carService.findCarByVin(vin);
}

/**
* GET Request - Receive all cars of a specific brand
* @param brand - Brand of the car
* @return - All cars with that brand
*/
@GetMapping("/cars/brands/{brand}")
public List<Car> findCarsByBrand(@PathVariable String brand) {
    return carService.findCarsByBrand(brand);
}

// and more...
```

```
/**
* POST Request - Input a Car instance
* @param carInput - The JSON body containing the required data
* @return - A new instance
*/
@PostMapping("/cars")
public Car insertCar(@RequestBody CarInput carInput) {
    return carService.insertCar(carInput);
}
```

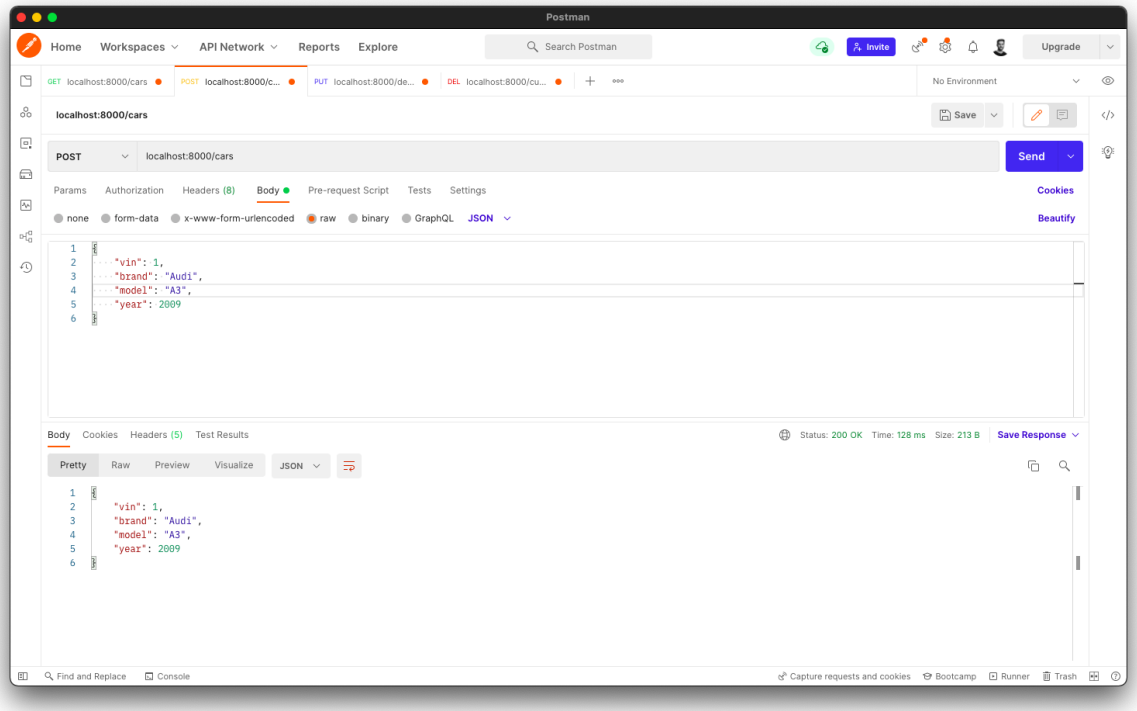
```
//**
* PUT Request - Update a Car instance
* @param carInput - The JSON body containing the updated data
* @param vin - VIN of the car, to identify which car we want to update
* @return - A updated instance with the updated data
*/
@PutMapping("/cars/{vin}")
public Car updateCar(@RequestBody CarInput carInput, @PathVariable Integer vin) {
```

```
    return carService.updateCar(carInput, vin);
}
```

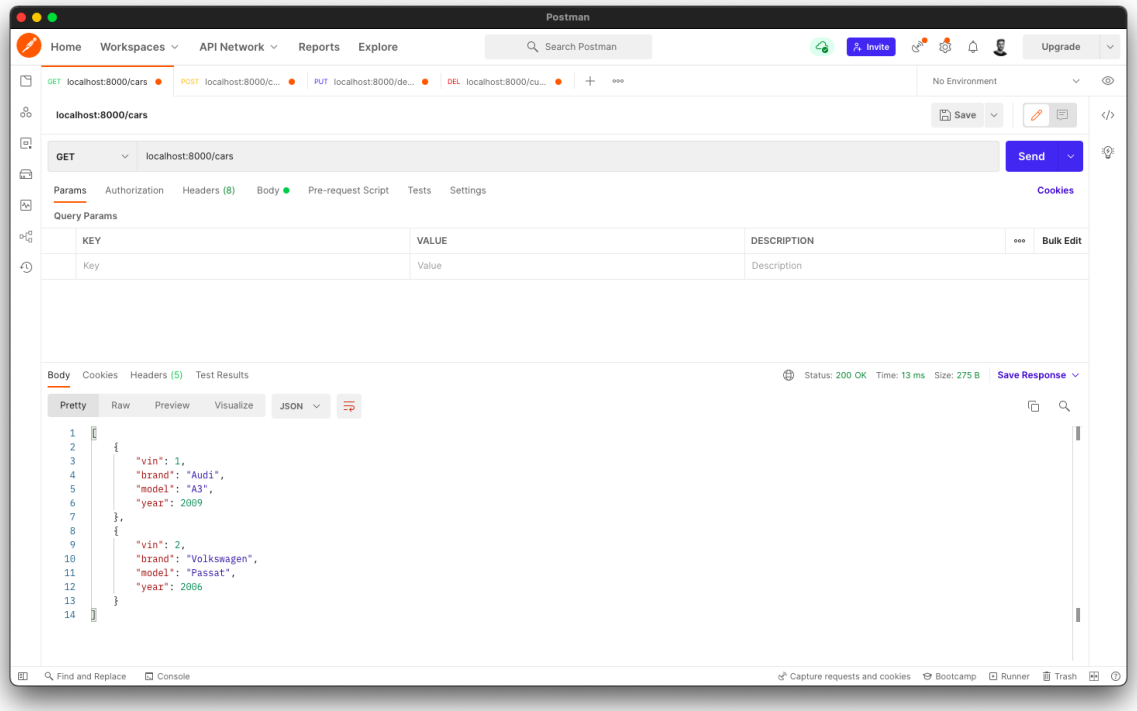
```
/**
* DELETE Request - Delete a Car instance
* @param vin - VIN of the car, to identify which car to delete
* @return - true if instance has been deleted
*/
@DeleteMapping("/cars/{vin}")
public Boolean deleteCar(@PathVariable Integer vin) {
    return carService.deleteCar(vin);
}
```

REST Examples

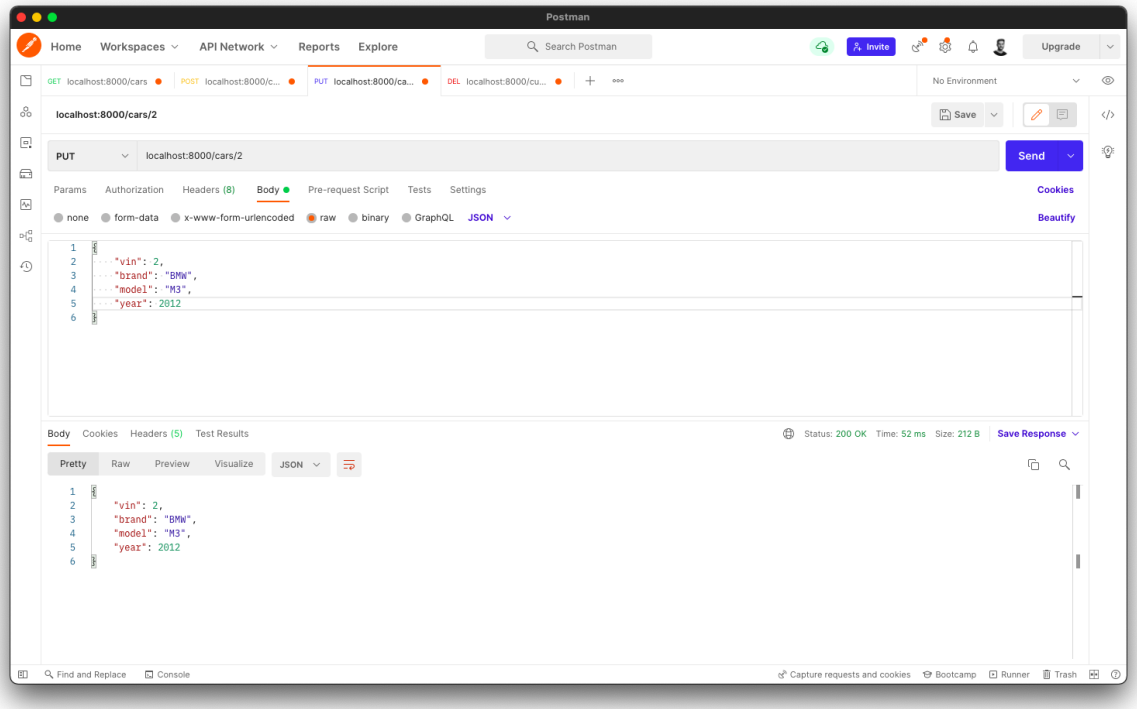
Here are some snapshots of using the various requests and their functionality in JSON:



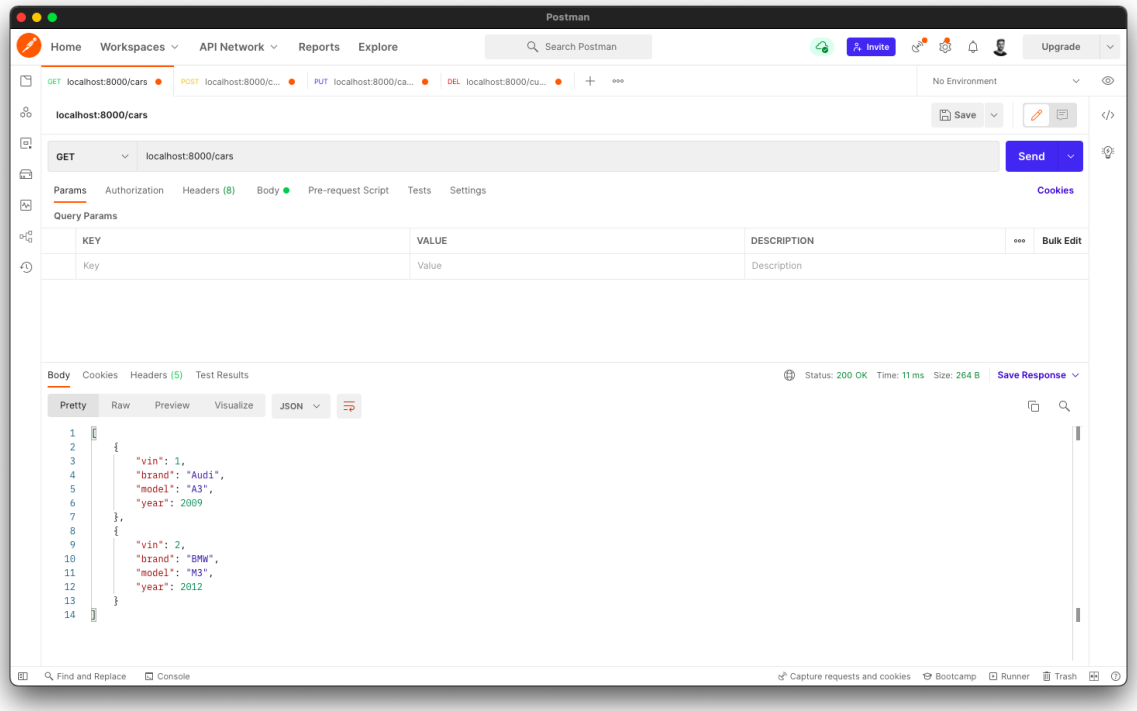
Here we have a POST request, where we create an instance of Car with the following data.



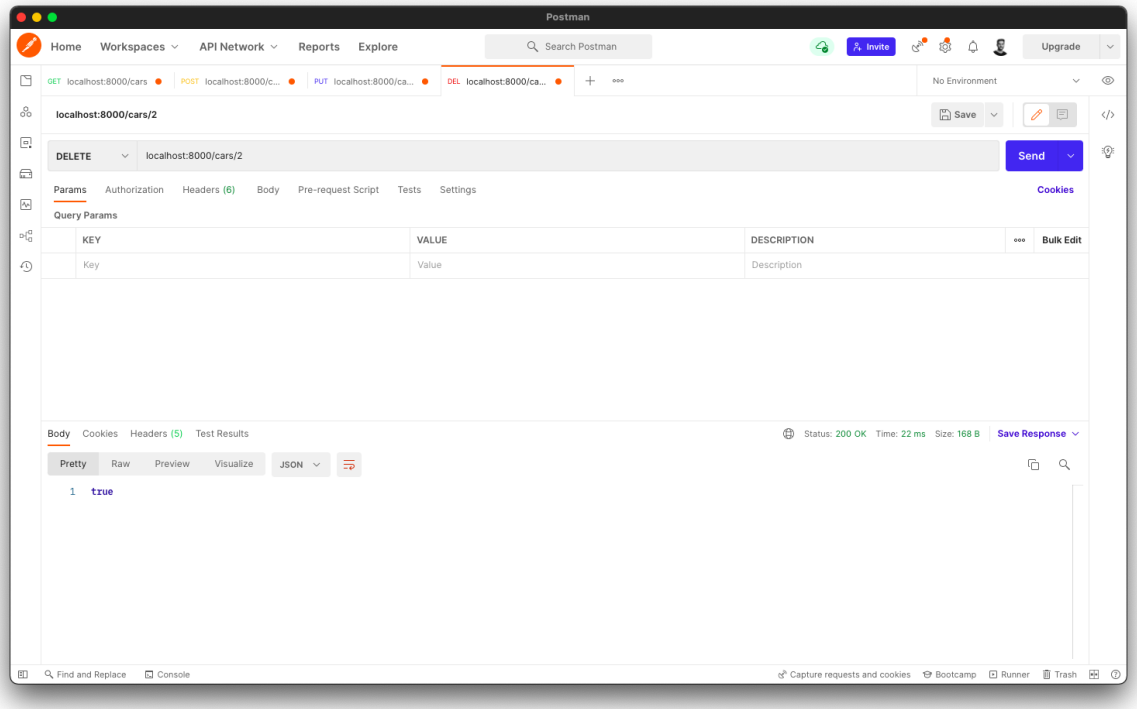
We use the GET request to receive all of the cars in the database along with their data.



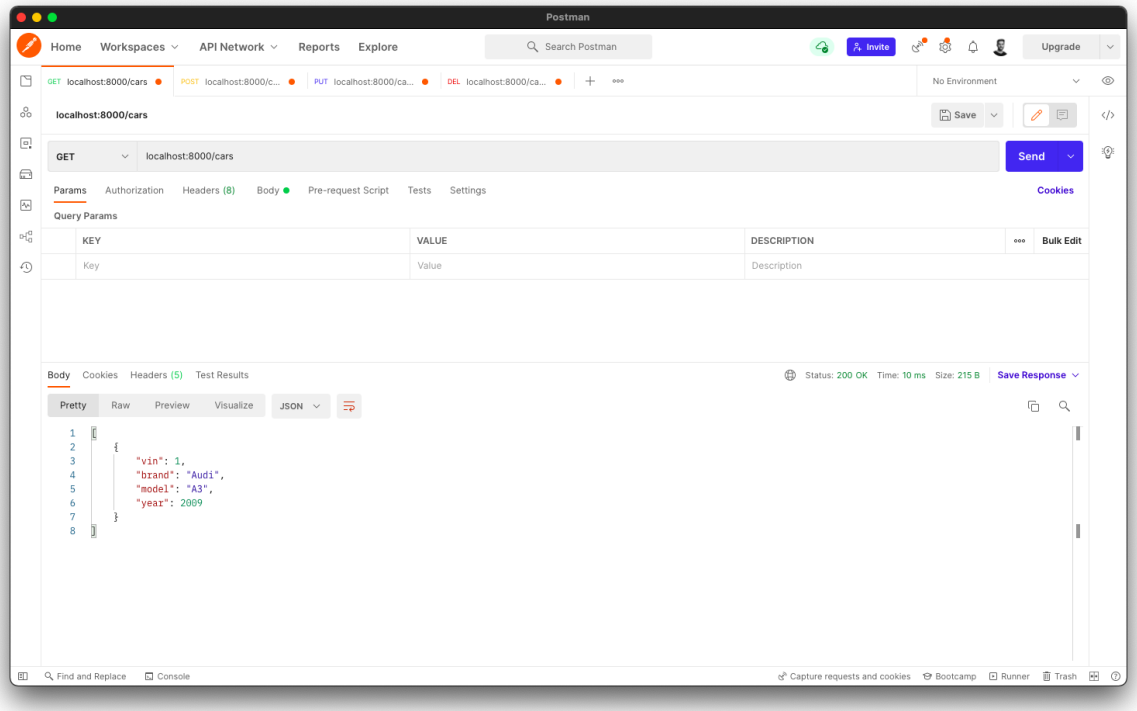
We use the PUT request to update an instance. In this case, we updated the car with the VIN: 2.



Here's a GET request after making a PUT request.



If we want to delete an instance, we use the DELETE request. If it's successful, true will be returned.



And here’s another GET request after using the DELETE request. Now we have only one instance.

JUnit Testing

For creating test methods, we have used the JUnit testing framework. In our case, we have to create methods to test if the saving, updating, and deleting features of our application work properly. As an addition, we implemented methods for each entity to test a single attribute of theirs. The methods should run in order and one-by-one (independently).

In total, we have 19 test methods that are crucial to determine if the application has no failures.

In order to have access to the database, we have to create “repo pointers” with the `@Autowired` annotation tag. Before every test method, we have to write the `@Test` annotation.

```
@Autowired
private CarRepository carRepository;

@Autowired
private CustomerRepository customerRepository;

@Autowired
private EmployeeRepository employeeRepository;
```

```
@Autowired
private DealershipRepository dealershipRepository;

@Autowired
private SaleRepository saleRepository;
```

testCreate() Test Method

```
@Test
@Order(1)
public void testCreateCar() {
    Car car = new Car(999, "Ford", "Mustang", 2018);
    carRepository.save(car);
    assertNotNull(carRepository.findCarByVin(999));
}
```

For the insertion of a car in the database, we do the following:

- Create a new Car object with the required data
- Save the object through `save()`
- Use `assertNotNull` to determine that the `findCarByVin` method isn't returning `NULL`.

This way, we can conclude that the Car object has been successfully inserted in the database.

testAttribute() Test Method

```
@Test
@Order(2)
public void testAttributeCar() {
    Car car = carRepository.findCarByVin(999);
    assertEquals(2018, car.getYear());
}
```

To see if an attribute of an entity has the correct value, we do the following:

- Obtain previously created Car object
- Use `assertEquals` to see if the value of an attribute is equal to the value we inputted

testUpdate() Test Method

```
@Test
@Order(3)
public void testUpdateCar() {
    Car car = carRepository.findCarByVin(999);
    car.setModel("F150");
    carRepository.save(car);
    assertNotEquals("Mustang", carRepository.findCarByVin(999).getModel());
}
```

To check if the data is correctly updated, we do the following:

- Obtain previously created Car object
- Change the value of the `model` with `setModel` → `“F150”`
- Save updated Car object
- Make sure that the current `model` value does not equal the previous `model` value

testDelete() Test Method

```
@Test
@Order(4)
public void testDeleteCar() {
    Car car = carRepository.findCarByVin(999);
    carRepository.delete(car);
    assertNull(carRepository.findCarByVin(999));
}
```

To test the delete method, we do the following:

- Obtain a Car object
- Call `delete()` method of `carRepository` and the Car object as a parameter
- Use the `assertNull` method to determine if the instance has been deleted

---

Summary

This is our first time creating a Spring Boot application with Maven and linking it to a MySQL database. It may be a simple project, but it helped us a lot to thoroughly understand a new way of building applications and we learned a new aspect of Java. Spring Boot can really ramp up your development pace and it allows you to test production-grade services from early-on.

We want to thank our professor, **Nuhi Besimi**, for teaching us the fundamentals of Java, the key features of Spring, but most importantly, for teaching us the discipline of a developer, the practices of one, and the principles of teamwork in development.

For any inquiries, you can contact:

- **Argjend Mehmedi** - am28507@seeu.edu.mk
- **Kan Misini** - km28487@seeu.edu.mk

Student Work

Argjend Mehmedi	Kan Misini
Creation, Properties, Dependencies, Organization	Database Creation & MySQL
Controller Implementation & REST Requests	Entity Classes
Default Services Implementation	Input Classes
Unit Testing	Repository Interfaces
Documentation	Service Interfaces
GitHub Repository / Pushing Code & Documents	

References

Spring Boot Documentation

Spring Annotations