Smart Car Monitoring Part 2

in previous chat Smart car monitoring system i upload some data there now i send remaining data here // Telemetry.java package com.smartcar.monitoring.model; import jakarta.persistence.*; import jakarta.validation.constraints.*; import java.time.LocalDateTime; @Entity @Table(name = "telemetry") public class Telemetry { @Id @GeneratedValue(strategy = GenerationType.IDENTITY) private Long id; @ManyToOne(fetch = FetchType.LAZY) @JoinColumn(name = "car_id", nullable = false) @NotNull(message = "Car is required") private Car car; @Column(nullable = false) @NotNull(message = "Timestamp is required") private LocalDateTime timestamp; @Column(nullable = false) @Min(value = 0, message = "Speed c. 4

// User.java package com.smartcar.monitoring.model; import jakarta.persistence.*; import jakarta.validation.constraints.*; import java.time.LocalDateTime; @Entity @Table(name = "users") public class User { @Id @GeneratedValue(strategy = GenerationType.IDENTITY) private Long id; @Column(unique = true, nullable = false) @NotBlank(message = "Username is required") @Size(min = 3, max = 50, message = "Username must be between 3 and 50 characters") private String username; @Column(nullable = false) @NotBlank(message = "Password is required") @Size(min = 6, message = "Password must be at least 6 characters") private String password; @Enumerated(EnumType.STRING) @Column(nullable = false). 8

// AdminRepository.java package com.smartcar.monitoring.repository; import com.smartcar.monitoring.model.Admin; import org.springframework.data.jpa.repository.JpaRepository; import org.springframework.data.jpa.repository.Query; import org.springframework.data.repository.query.Param; import org.springframework.stereotype.Repository; import java.util.List; import java.util.Optional; @Repository public interface AdminRepository extends JpaRepository<Admin, Long> { // Find by user ID Optional<Admin> findByUserId(Long userId); // Find all active admins List<Admin> findByIsActiveTrue(); // Find admins by permissions List<Admin> findByPermissionsContainingAndIsActiveTrue(String permission); // Find admins created. 15

// AlertRepository.java package com.smartcar.monitoring.repository; import com.smartcar.monitoring.model.Alert; import com.smartcar.monitoring.model.Alert.AlertSeverity; import org.springframework.data.jpa.repository.JpaRepository; import org.springframework.data.jpa.repository.Query; import org.springframework.data.repository.query.Param; import org.springframework.stereotype.Repository; import java.time.LocalDateTime; import java.util.List; @Repository public interface AlertRepository extends JpaRepository<Alert, Long> { // Find by car ID List<Alert> findByCarIdAndIsActiveTrue(Long carId); // Find by alert type List<Alert> findByTypeAndIsActiveTrue(String type); // Find by severity List<Alert> findBySeverityA. 17

// CarRepository.java package com.smartcar.monitoring.repository; import com.smartcar.monitoring.model.Car; import org.springframework.data.jpa.repository.JpaRepository; import org.springframework.data.jpa.repository.Query; import org.springframework.data.repository.query.Param; import org.springframework.stereotype.Repository; import java.util.List; import java.util.Optional; @Repository public interface CarRepository extends JpaRepository<Car, Long> { // Find by driver ID List<Car> findByDriverIdAndIsActiveTrue(Long driverId); // Find by status List<Car> findByStatusAndIsActiveTrue(String status); // Find cars by fuel level threshold List<Car> findByFuelLevelLessThanAndIsActiveTrue(Integer fuelThreshold);. 20

// DriverRepository.java package com.smartcar.monitoring.repository; import com.smartcar.monitoring.model.Driver; import org.springframework.data.jpa.repository.JpaRepository; import org.springframework.data.jpa.repository.Query; import org.springframework.data.repository.query.Param; import org.springframework.stereotype.Repository; import java.util.List; import java.util.Optional; @Repository public interface DriverRepository extends JpaRepository<Driver, Long> { // Find by user ID Optional<Driver> findByUserId(Long userId); // Find by assigned car ID Optional<Driver> findByAssignedCarId(Long carId); // Find all active drivers List<Driver> findByIsActiveTrue(); // Find drivers without assigned cars L. 23

// TelemetryRepository.java package com.smartcar.monitoring.repository; import com.smartcar.monitoring.model.Telemetry; import org.springframework.data.jpa.repository.JpaRepository; import org.springframework.data.jpa.repository.Query; import org.springframework.data.repository.query.Param; import org.springframework.stereotype.Repository; import java.time.LocalDateTime; import java.util.List; @Repository public interface TelemetryRepository extends JpaRepository<Telemetry, Long> { // Find by car ID List<Telemetry> findByCarIdAndIsActiveTrue(Long carId); // Find by timestamp range List<Telemetry> findByTimestampBetweenAndIsActiveTrue(LocalDateTime startTime, LocalDateTime endTime); // Find by car ID and timestamp r. 25

// UserRepository.java package com.smartcar.monitoring.repository; import com.smartcar.monitoring.model.User; import com.smartcar.monitoring.model.User.UserRole; import org.springframework.data.jpa.repository.JpaRepository; import org.springframework.data.jpa.repository.Query; import org.springframework.data.repository.query.Param; import org.springframework.stereotype.Repository; import java.util.List; import java.util.Optional; @Repository public interface UserRepository extends JpaRepository<User, Long> { // Find by username (for authentication) Optional<User> findByUsername(String username); // Find by email Optional<User> findByEmail(String email); // Find by role List<User> findByRole(UserRole role); 28

// AdminService.java package com.smartcar.monitoring.service; import com.smartcar.monitoring.model.Admin; import com.smartcar.monitoring.model.User; import com.smartcar.monitoring.repository.AdminRepository; import com.smartcar.monitoring.repository.UserRepository; import com.smartcar.monitoring.exception.AdminNotFoundException; import com.smartcar.monitoring.exception.UserNotFoundException; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Transactional; import java.time.LocalDateTime; import java.util.List; import java.util.Optional; @Service @Transactional public class AdminService { @Autowired private AdminRepository. 31

// AlertService.java package com.smartcar.monitoring.service; import com.smartcar.monitoring.model.Alert; import com.smartcar.monitoring.model.Car; import com.smartcar.monitoring.model.Alert.AlertSeverity; import com.smartcar.monitoring.repository.AlertRepository; import com.smartcar.monitoring.exception.AlertNotFoundException; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Transactional; import java.time.LocalDateTime; import java.util.List; @Service @Transactional public class AlertService { @Autowired private AlertRepository alertRepository; // Create new alert public Alert createAlert(Car car, String type. 37

// CarService.java package com.smartcar.monitoring.service; import com.smartcar.monitoring.model.Car; import com.smartcar.monitoring.model.Driver; import com.smartcar.monitoring.model.Telemetry; import com.smartcar.monitoring.repository.CarRepository; import com.smartcar.monitoring.repository.DriverRepository; import com.smartcar.monitoring.repository.TelemetryRepository; import com.smartcar.monitoring.exception.CarNotFoundException; import com.smartcar.monitoring.exception.DriverNotFoundException; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Transactional; import java.time.LocalDateTime; import java.util.List; import java.ut. 42

// DriverService.java package com.smartcar.monitoring.service; import com.smartcar.monitoring.model.Driver; import com.smartcar.monitoring.model.User; import com.smartcar.monitoring.repository.DriverRepository; import com.smartcar.monitoring.repository.UserRepository; import com.smartcar.monitoring.exception.DriverNotFoundException; import com.smartcar.monitoring.exception.UserNotFoundException; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Transactional; import java.time.LocalDateTime; import java.util.List; import java.util.Optional; @Service @Transactional public class DriverService { @Autowired private DriverRepo. 49

// MqttService.java package com.smartcar.monitoring.service; import com.fasterxml.jackson.databind.ObjectMapper; import com.smartcar.monitoring.dto.TelemetryDto; import com.smartcar.monitoring.model.Alert; import com.smartcar.monitoring.model.Car; import com.smartcar.monitoring.model.Telemetry; import org.eclipse.paho.client.mqttv3.*; import org.slf4j.Logger; import org.slf4j.LoggerFactory; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.beans.factory.annotation.Value; import org.springframework.stereotype.Service; import jakarta.annotation.PostConstruct; import jakarta.annotation.PreDestroy; import java.time.LocalDateTime; import java.util.HashMap; import java.util.Map; import java.util.concurren. 54

// TelemetryService.java package com.smartcar.monitoring.service; import com.smartcar.monitoring.model.Telemetry; import com.smartcar.monitoring.model.Car; import com.smartcar.monitoring.repository.TelemetryRepository; import com.smartcar.monitoring.repository.CarRepository; import com.smartcar.monitoring.exception.TelemetryNotFoundException; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Transactional; import java.time.LocalDateTime; import java.util.List; @Service @Transactional public class TelemetryService { @Autowired private TelemetryRepository telemetryRepository; @Autowired private CarRepository carRe. 62

// UserService.java package com.smartcar.monitoring.service; import com.smartcar.monitoring.model.User; import com.smartcar.monitoring.model.User.UserRole; import com.smartcar.monitoring.repository.UserRepository; import com.smartcar.monitoring.exception.UserNotFoundException; import com.smartcar.monitoring.exception.UserAlreadyExistsException; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.security.crypto.password.PasswordEncoder; import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Transactional; import java.time.LocalDateTime; import java.util.List; import java.util.Optional; @Service @Transactional public class UserService { @Autowired pri. 70

// WebSocketService.java package com.smartcar.monitoring.service; import com.fasterxml.jackson.databind.ObjectMapper; import com.smartcar.monitoring.dto.TelemetryDto; import com.smartcar.monitoring.model.Alert; import org.slf4j.Logger; import org.slf4j.LoggerFactory; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.messaging.simp.SimpMessagingTemplate; import org.springframework.stereotype.Service; import java.time.LocalDateTime; import java.util.HashMap; import java.util.Map; @Service public class WebSocketService { private static final Logger logger = LoggerFactory.getLogger(WebSocketService.class); @Autowired private SimpMessagingTemplate messagingTemplate; @Autowired pri. 75

// TelemetrySimulator.java package com.smartcar.monitoring.simulator; import com.smartcar.monitoring.dto.TelemetryDto; import com.smartcar.monitoring.model.Car; import com.smartcar.monitoring.service.CarService; import com.smartcar.monitoring.service.MqttService; import com.smartcar.monitoring.service.WebSocketService; import org.slf4j.Logger; import org.slf4j.LoggerFactory; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.beans.factory.annotation.Value; import org.springframework.scheduling.annotation.Scheduled; import org.springframework.stereotype.Component; import jakarta.annotation.PostConstruct; import java.time.LocalDateTime; import java.util.List; import java.util.Random; import java.util.c. 81

# application.properties spring.application.name=smart-car-monitoring # Server Configuration server.port=8080 # Database Configuration spring.datasource.url=jdbc:mysql://localhost:3306/smartcar spring.datasource.username=root spring.datasource.password=1234 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver # JPA Configuration spring.jpa.hibernate.ddl-auto=update spring.jpa.show-sql=true spring.jpa.properties.hibernate.format_sql=true spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect # JWT Configuration jwt.secret=smartCarMonitoringSecretKey2025ForJWTTokenGeneration jwt.expiration=86400000 # MQTT Configuration - HiveMQ Cloud mqtt.broker.url=ssl://28f28b09486648cc85d19ad078bcfe44.s1.eu.hivemq.cloud:888. . . 87

in previous chat Smart car monitoring system i upload some data there now i send remaining data here

```
// Telemetry.java
package com.smartcar.monitoring.model;

import jakarta.persistence.*;
import jakarta.validation.constraints.*;
import java.time.LocalDateTime;

@Entity
@Table(name = "telemetry")
public class Telemetry {

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

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "car_id", nullable = false)
    @NotNull(message = "Car is required")
    private Car car;

    @Column(nullable = false)
    @NotNull(message = "Timestamp is required")
    private LocalDateTime timestamp;

    @Column(nullable = false)
    @Min(value = 0, message = "Speed cannot be negative")
    @Max(value = 200, message = "Speed cannot exceed 200 km/h")
    private Integer speed;

    @Column(nullable = false)
    @Min(value = 0, message = "Fuel cannot be negative")
    @Max(value = 100, message = "Fuel cannot exceed 100%")
    private Integer fuel;

    @Column(nullable = false)
    @Min(value = -20, message = "Temperature cannot be below -20°C")
    @Max(value = 60, message = "Temperature cannot exceed 60°C")
    private Integer temperature;

    @Column(nullable = false)
    @NotBlank(message = "Location is required")
    private String location;
    @Column(name = "creation_date", nullable = false)
    private LocalDateTime creationDate;

    @Column(name = "last_update_on")
    private LocalDateTime lastUpdateOn;

    @Column(name = "is_active", nullable = false)
    private Boolean isActive = true;

    // Constructors
    public Telemetry() {
        this.creationDate = LocalDateTime.now();
        this.lastUpdateOn = LocalDateTime.now();
        this.isActive = true;
        this.timestamp = LocalDateTime.now();
    }

    public Telemetry(Car car, Integer speed, Integer fuel, Integer temperature, String location) {
        this();
        this.car = car;
        this.speed = speed;
        this.fuel = fuel;
        this.temperature = temperature;
        this.location = location;
    }

    // Getters and Setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public Car getCar() {
        return car;
    }

    public void setCar(Car car) {
        this.car = car;
    }

    public LocalDateTime getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(LocalDateTime timestamp) {
        this.timestamp = timestamp;
    }
    public Integer getSpeed() {
        return speed;
    }

    public void setSpeed(Integer speed) {
        this.speed = speed;
    }

    public Integer getFuel() {
        return fuel;
    }

    public void setFuel(Integer fuel) {
        this.fuel = fuel;
    }

    public Integer getTemperature() {
        return temperature;
    }

    public void setTemperature(Integer temperature) {
        this.temperature = temperature;
    }

    public String getLocation() {
        return location;
    }

    public void setLocation(String location) {
        this.location = location;
    }

    public LocalDateTime getCreationDate() {
        return creationDate;
    }

    public void setCreationDate(LocalDateTime creationDate) {
        this.creationDate = creationDate;
    }

    public LocalDateTime getLastUpdateOn() {
        return lastUpdateOn;
    }

    public void setLastUpdateOn(LocalDateTime lastUpdateOn) {
        this.lastUpdateOn = lastUpdateOn;
    }

    public Boolean getIsActive() {
        return isActive;
    }
    public void setIsActive(Boolean isActive) {
        this.isActive = isActive;
    }

    // Pre-update hook
    @PreUpdate
    public void preUpdate() {
        this.lastUpdateOn = LocalDateTime.now();
    }

    @Override
    public String toString() {
        return "Telemetry{" +
                "id=" + id +
                ", carId=" + (car != null ? car.getId() : null) +
                ", timestamp=" + timestamp +
                ", speed=" + speed +
                ", fuel=" + fuel +
                ", temperature=" + temperature +
                ", location='" + location + '\'' +
                ", isActive=" + isActive +
                '}';
    }
}
```

Perfect! Thanks for sharing the `Telemetry.java` entity. From what I can see, this entity is well-structured for your **Smart Car Monitoring System**. Let me break down and give a few observations and potential improvements:

---

1. Entity Structure

- `Telemetry` is linked to `Car` with a **Many-to-One** relationship.
- Includes fields for:
    - `timestamp`, `speed`, `fuel`, `temperature`, `location`
    - Audit fields: `creationDate`, `lastUpdateOn`, `isActive`
- Uses JPA and Jakarta validation annotations properly.

---

2. Validations

- `@Min` and `@Max` constraints on speed, fuel, and temperature are realistic for a car telemetry system.
- `@NotBlank` for `location` ensures it's always populated.
- Good use of `@NotNull` for required fields.

---

3. Constructors

- Default constructor sets `creationDate`, `lastUpdateOn`, `isActive`, and `timestamp` to `now()`.
- Parameterized constructor delegates to default constructor, which is a clean approach.

---

4. Lifecycle Hook

- `@PreUpdate` updates `lastUpdateOn` automatically, which is perfect for tracking updates.

---

5. `toString()` Method

- Prints `carId` instead of the full `Car` object to avoid lazy-loading issues. Smart move!

---

6. Suggestions / Enhancements

1. **Timestamp Setting**
   - Currently, both `creationDate` and `timestamp` are set to `LocalDateTime.now()` in the default constructor.
     If you want telemetry to represent actual car time, consider setting `timestamp` externally when a new reading is received rather than automatically to now.
2. **`@PrePersist` Hook**
   - Optionally, you can use `@PrePersist` to set `creationDate` and `lastUpdateOn` instead of in constructor:

```java
@PrePersist
public void prePersist() {
    this.creationDate = LocalDateTime.now();
    this.lastUpdateOn = LocalDateTime.now();
}
```

3. **Optional Optimizations**
   - Add `@Column(length = 100)` for `location` if you want to limit string size.
   - Consider adding `@Index` on `timestamp` for faster querying of telemetry data over time.

Overall, this entity looks solid and production-ready for your system.

If you want, I can also **show how to write the `TelemetryRepository` and a simple service to save/read telemetry data efficiently**, which will make this entity plug-and-play in your Spring Boot app.

Do you want me to do that next?

```
// User.java
package com.smartcar.monitoring.model;

import jakarta.persistence.*;
import jakarta.validation.constraints.*;
import java.time.LocalDateTime;

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

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

    @Column(unique = true, nullable = false)
    @NotBlank(message = "Username is required")
    @Size(min = 3, max = 50, message = "Username must be between 3 and 50 characters")
    private String username;

    @Column(nullable = false)
    @NotBlank(message = "Password is required")
    @Size(min = 6, message = "Password must be at least 6 characters")
    private String password;

    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    @NotNull(message = "Role is required")
    private UserRole role;

    @Column(nullable = false)
    @NotBlank(message = "Name is required")
    @Size(max = 100, message = "Name cannot exceed 100 characters")
    private String name;

    @Column(nullable = false)
    @Min(value = 18, message = "Age must be at least 18")
    @Max(value = 100, message = "Age cannot exceed 100")
    private Integer age;

    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    @NotNull(message = "Gender is required")
    private Gender gender;

    @Column(name = "contact_number", nullable = false)
    @NotBlank(message = "Contact number is required")
    @Pattern(regexp = "^[+]?[0-9]{10,15}$", message = "Invalid contact number format")
    private String contactNumber;

    @Column(unique = true, nullable = false)
    @Email(message = "Invalid email format")
    @NotBlank(message = "Email is required")
    private String email;
    @Column(name = "license_number", nullable = false)
    @NotBlank(message = "License number is required")
    private String licenseNumber;

    @Column(name = "creation_date", nullable = false)
    private LocalDateTime creationDate;

    @Column(name = "last_update_on")
    private LocalDateTime lastUpdateOn;

    @Column(name = "is_active", nullable = false)
    private Boolean isActive = true;

    // Enums
    public enum UserRole {
        ADMIN, DRIVER
    }

    public enum Gender {
        MALE, FEMALE, OTHER
    }

    // Constructors
    public User() {
        this.creationDate = LocalDateTime.now();
        this.lastUpdateOn = LocalDateTime.now();
        this.isActive = true;
    }

    public User(String username, String password, UserRole role, String name, Integer age,
            Gender gender, String contactNumber, String email, String licenseNumber) {
        this();
        this.username = username;
        this.password = password;
        this.role = role;
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.contactNumber = contactNumber;
        this.email = email;
        this.licenseNumber = licenseNumber;
    }

    // Getters and Setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public UserRole getRole() {
        return role;
    }

    public void setRole(UserRole role) {
        this.role = role;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public Gender getGender() {
        return gender;
    }

    public void setGender(Gender gender) {
        this.gender = gender;
    }

    public String getContactNumber() {
        return contactNumber;
    }

    public void setContactNumber(String contactNumber) {
        this.contactNumber = contactNumber;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getLicenseNumber() {
        return licenseNumber;
    }

    public void setLicenseNumber(String licenseNumber) {
        this.licenseNumber = licenseNumber;
    }

    public LocalDateTime getCreationDate() {
        return creationDate;
    }

    public void setCreationDate(LocalDateTime creationDate) {
        this.creationDate = creationDate;
    }

    public LocalDateTime getLastUpdateOn() {
        return lastUpdateOn;
    }

    public void setLastUpdateOn(LocalDateTime lastUpdateOn) {
        this.lastUpdateOn = lastUpdateOn;
    }

    public Boolean getIsActive() {
        return isActive;
    }

    public void setIsActive(Boolean isActive) {
        this.isActive = isActive;
    }

    // Pre-update hook
    @PreUpdate
    public void preUpdate() {
        this.lastUpdateOn = LocalDateTime.now();
    }
    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", role=" + role +
                ", name='" + name + '\'' +
                ", email='" + email + '\'' +
                ", licenseNumber='" + licenseNumber + '\'' +
                ", isActive=" + isActive +
                '}';
    }
}
```

Awesome! Thanks for sharing the `User.java` entity. Let's go through it carefully and see how it fits into your **Smart Car Monitoring System**.

---

1. Entity Structure

- The `User` entity is mapped to a table `users`.
- It contains typical fields:
  - Authentication: `username`, `password`
  - Profile: `name`, `age`, `gender`, `contactNumber`, `email`, `licenseNumber`
  - Role: `UserRole` enum (`ADMIN` / `DRIVER`)
  - Status tracking: `creationDate`, `lastUpdateOn`, `isActive`
- Validation annotations (`@NotBlank`, `@Email`, `@Pattern`, `@Size`, `@Min`/`@Max`) are well-applied.

---

2. Enum Usage

- `UserRole` and `Gender` enums are stored as strings in the DB (`@Enumerated(EnumType.STRING)`)—good choice for readability.

---

3. Constructors

- Default constructor initializes `creationDate`, `lastUpdateOn`, and `isActive` correctly.
- Parameterized constructor delegates to default constructor for DRY code—clean implementation.

---

4. Lifecycle Hook

- `@PreUpdate` ensures `lastUpdateOn` is updated automatically whenever the entity is updated.

---

5. `toString()` Method

- Only prints selected fields (`id`, `username`, `role`, `name`, `email`, `licenseNumber`, `isActive`) to avoid sensitive data like `password`—excellent practice.

---

6. Suggestions / Improvements

1. **Password Handling**
    - Store hashed passwords instead of plain text for security.
    - You can use `BCryptPasswordEncoder` in Spring Security:

```java
public void setPassword(String password) {
    this.password = new BCryptPasswordEncoder().encode(password);
}
```

2. **Phone Number Validation**
    - Current regex allows `+` and 10–15 digits which is good.
    - Optional: enforce only digits after optional `+`.
3. **`@PrePersist` Hook**
    - Like `Telemetry`, consider using `@PrePersist` to set creation and last update timestamps:

```java
@PrePersist
public void prePersist() {
    this.creationDate = LocalDateTime.now();
    this.lastUpdateOn = LocalDateTime.now();
}
```

4. **Indexing**
    - Add DB indices for `username` and `email` for faster lookups:

```java
@Table(name = "users", indexes = {
    @Index(name = "idx_username", columnList = "username"),
    @Index(name = "idx_email", columnList = "email")
})
```

5. **Relation with `Driver` / `Car`**
    - If a user can be a driver of a car, you might want to add a `@OneToOne` or `@OneToMany` mapping to the `Driver` entity (which you shared previously).
      Example:

```java
@OneToOne(mappedBy = "user")
private Driver driver;
```

Overall, this entity is **robust, clean, and ready for production**, with just minor improvements for security and performance.

If you want, I can **map out the complete relationship diagram between `User`, `Driver`, `Car`, and `Telemetry`**, so we have a full picture of your system before writing repositories and services.

Do you want me to do that next?

```
// AdminRepository.java
package com.smartcar.monitoring.repository;

import com.smartcar.monitoring.model.Admin;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Optional;

@Repository
public interface AdminRepository extends JpaRepository<Admin, Long> {

    // Find by user ID
    Optional<Admin> findByUserId(Long userId);

    // Find all active admins
    List<Admin> findByIsActiveTrue();

    // Find admins by permissions
    List<Admin> findByPermissionsContainingAndIsActiveTrue(String permission);

    // Find admins created in date range
    @Query("SELECT a FROM Admin a WHERE a.creationDate BETWEEN :startDate AND :endDate
AND a.isActive = true")
    List<Admin> findAdminsCreatedBetween(@Param("startDate") java.time.LocalDateTime
startDate,
            @Param("endDate") java.time.LocalDateTime endDate);

    // Count active admins
    long countByIsActiveTrue();

    // Count admins with specific permission
    long countByPermissionsContainingAndIsActiveTrue(String permission);
}
```

Perfect! Your **`AdminRepository.java`** is well-written and leverages Spring Data JPA features effectively. Here's a breakdown and some suggestions:

1. Repository Purpose

- Extends `JpaRepository<Admin, Long>` → provides standard CRUD operations out-of-the-box.
- Includes **custom queries** to handle common admin-related operations:
  - Find by user ID
  - Get all active admins
  - Filter by permissions
  - Filter by creation date range
  - Count operations

---

2. Methods Overview

1. `findByUserId(Long userId)`
   - Returns an `Optional<Admin>`. Good for handling non-existent IDs safely.
2. `findByIsActiveTrue()`
   - Fetches all active admins only. Clean and readable.
3. `findByPermissionsContainingAndIsActiveTrue(String permission)`
   - Filters active admins with a specific permission. Efficient if permissions are stored as a delimited string.
4. `findAdminsCreatedBetween(startDate, endDate)`
   - Uses JPQL query with `@Query` annotation.
   - Ensures results are only active admins within a date range.
5. `countByIsActiveTrue()`
   - Counts active admins. Direct method naming convention of Spring Data JPA.
6. `countByPermissionsContainingAndIsActiveTrue(String permission)`
   - Counts admins with a specific permission and active status.

---

3. Suggestions / Enhancements

1. **Permissions Storage**
   - Right now, it seems like `permissions` is a string containing multiple values.
   - Consider changing it to a `@ElementCollection` **or a separate `Permission` entity** for cleaner queries and better performance.

java

```java
@ElementCollection(fetch = FetchType.EAGER)
@CollectionTable(name = "admin_permissions", joinColumns = @JoinColumn(name = "admin_id"))
@Column(name = "permission")
private Set<String> permissions;
```

2. **Optional Optimizations**
    - For queries like `findByPermissionsContainingAndIsActiveTrue`, adding a **database index** on the permissions column could improve performance if dataset is large.
3. **Date Range Query**
    - The current JPQL query works, but if you want flexibility for filtering by `creationDate` or `lastUpdateOn`, you could add an optional parameter for the latter.

This repository is **solid and production-ready** for your admin operations.

If you want, I can also **write a `UserRepository` and `TelemetryRepository` in a similar style** so all major entities have clean JPA access layers. This will help you integrate services easily.

Do you want me to do that next?

```
// AlertRepository.java
package com.smartcar.monitoring.repository;

import com.smartcar.monitoring.model.Alert;
import com.smartcar.monitoring.model.Alert.AlertSeverity;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.time.LocalDateTime;
import java.util.List;

@Repository
public interface AlertRepository extends JpaRepository<Alert, Long> {

    // Find by car ID
    List<Alert> findByCarIdAndIsActiveTrue(Long carId);

    // Find by alert type
    List<Alert> findByTypeAndIsActiveTrue(String type);

    // Find by severity
    List<Alert> findBySeverityAndIsActiveTrue(AlertSeverity severity);

    // Find by acknowledgment status
    List<Alert> findByAcknowledgedAndIsActiveTrue(Boolean acknowledged);

    // Find by timestamp range
    List<Alert> findByTimestampBetweenAndIsActiveTrue(LocalDateTime startTime, LocalDateTime endTime);

    // Find by car ID and timestamp range
    List<Alert> findByCarIdAndTimestampBetweenAndIsActiveTrue(Long carId, LocalDateTime
startTime,
            LocalDateTime endTime);

    // Find unacknowledged alerts
    List<Alert> findByAcknowledgedFalseAndIsActiveTrue();

    // Find unacknowledged alerts by car
    List<Alert> findByCarIdAndAcknowledgedFalseAndIsActiveTrue(Long carId);

    // Find unacknowledged alerts by severity
    List<Alert> findBySeverityAndAcknowledgedFalseAndIsActiveTrue(AlertSeverity severity);

    // Find all active alerts
    List<Alert> findByIsActiveTrue();

    // Find alerts created in date range
    @Query("SELECT a FROM Alert a WHERE a.creationDate BETWEEN :startDate AND :endDate
AND a.isActive = true")
    List<Alert> findAlertsCreatedBetween(@Param("startDate") LocalDateTime startDate,
            @Param("endDate") LocalDateTime endDate);

    // Find critical alerts (HIGH and CRITICAL severity)
    @Query("SELECT a FROM Alert a WHERE a.severity IN ('HIGH', 'CRITICAL') AND a.isActive =
true")
    List<Alert> findCriticalAlerts();

    // Find alerts by car and severity
    List<Alert> findByCarIdAndSeverityAndIsActiveTrue(Long carId, AlertSeverity severity);

    // Find alerts by car and type
    List<Alert> findByCarIdAndTypeAndIsActiveTrue(Long carId, String type);

    // Count alerts by car
    long countByCarIdAndIsActiveTrue(Long carId);

    // Count total active alerts
    long countByIsActiveTrue();

    // Count unacknowledged alerts
    long countByAcknowledgedFalseAndIsActiveTrue();

    // Count alerts by severity
    long countBySeverityAndIsActiveTrue(AlertSeverity severity);

    // Count alerts by type
    long countByTypeAndIsActiveTrue(String type);

    // Count alerts in time range
    long countByTimestampBetweenAndIsActiveTrue(LocalDateTime startTime, LocalDateTime
endTime);
```

```
    // Count critical alerts
    long countBySeverityInAndIsActiveTrue(List<AlertSeverity> severities);
}
```

Perfect! Your `AlertRepository.java` is very comprehensive and covers almost all the queries you'd need for alert management in a **Smart Car Monitoring System**. Let's break it down:

---

1. Repository Purpose

- Extends `JpaRepository<Alert, Long>` → gives standard CRUD operations.
- Adds **custom queries** to filter alerts by:
    - Car ID
    - Type
    - Severity
    - Acknowledgment status
    - Timestamp / date ranges
- Counts alerts based on different criteria.
- Fetches "critical" alerts (HIGH or CRITICAL severity).

---

2. Observations

1. **Find Methods**
    - Methods like `findByCarIdAndTimestampBetweenAndIsActiveTrue` and `findBySeverityAndAcknowledgedFalseAndIsActiveTrue` are great for filtering active alerts efficiently.
    - `findCriticalAlerts()` uses JPQL which is correct, though you could also use a list parameter for flexibility.
2. **Count Methods**
    - Provides counts by car, severity, type, acknowledgment, time range, and total active alerts.
    - Very useful for dashboards and statistics.
3. **Timestamp Queries**
    - Queries like `findByTimestampBetweenAndIsActiveTrue` are perfect for monitoring alerts over time.
4. **Unacknowledged Alerts**
    - `findByAcknowledgedFalseAndIsActiveTrue` and its variations make it easy to track pending alerts.

---

3. Suggestions / Enhancements

1. **Critical Alerts Flexibility**
    - Instead of hardcoding `'HIGH', 'CRITICAL'`, you could pass a list of severities:

```java
@Query("SELECT a FROM Alert a WHERE a.severity IN :severities AND a.isActive = true")
List<Alert> findAlertsBySeverities(@Param("severities") List<AlertSeverity> severities);
```

This makes it reusable for any set of severities.

2. **Indexes**
    - Consider DB indices on `carId`, `timestamp`, `severity`, and `acknowledged` to improve query performance if your alerts table grows large.
3. **Optional Pagination**
    - For large datasets, consider `Pageable` support for methods like:

```java
List<Alert> findByCarIdAndIsActiveTrue(Long carId, Pageable pageable);
```

4. **Soft Delete Consistency**
    - All queries check `isActive = true`, which is consistent with soft-delete pattern. Nice.

Overall, this repository is **robust and production-ready**, covering almost every scenario you'd need in alert tracking and analytics.

If you want, I can now **draw a complete entity-relationship map** showing `User`, `Driver`, `Car`, `Telemetry`, `Alert`, and `Admin` with their relationships, so your system is fully visualized.

Do you want me to do that?

```
// CarRepository.java
package com.smartcar.monitoring.repository;

import com.smartcar.monitoring.model.Car;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Optional;

@Repository
public interface CarRepository extends JpaRepository<Car, Long> {

    // Find by driver ID
    List<Car> findByDriverIdAndIsActiveTrue(Long driverId);

    // Find by status
    List<Car> findByStatusAndIsActiveTrue(String status);
    // Find cars by fuel level threshold
    List<Car> findByFuelLevelLessThanAndIsActiveTrue(Integer fuelThreshold);

    // Find cars by temperature range
    List<Car> findByTemperatureBetweenAndIsActiveTrue(Integer minTemp, Integer maxTemp);

    // Find cars by speed range
    List<Car> findBySpeedBetweenAndIsActiveTrue(Integer minSpeed, Integer maxSpeed);

    // Find cars by location
    List<Car> findByLocationContainingAndIsActiveTrue(String location);

    // Find all active cars
    List<Car> findByIsActiveTrue();

    // Find cars without assigned drivers
    List<Car> findByDriverIsNullAndIsActiveTrue();

    // Find cars with assigned drivers
    List<Car> findByDriverIsNotNullAndIsActiveTrue();

    // Find cars updated in time range (for real-time monitoring)
    @Query("SELECT c FROM Car c WHERE c.lastUpdateOn >= :since AND c.isActive = true")
    List<Car> findCarsUpdatedSince(@Param("since") java.time.LocalDateTime since);

    // Find cars by creation date range
    @Query("SELECT c FROM Car c WHERE c.creationDate BETWEEN :startDate AND :endDate
AND c.isActive = true")
    List<Car> findCarsCreatedBetween(@Param("startDate") java.time.LocalDateTime startDate,
            @Param("endDate") java.time.LocalDateTime endDate);

    // Count active cars
    long countByIsActiveTrue();

    // Count cars by status
    long countByStatusAndIsActiveTrue(String status);

    // Count cars with low fuel
    long countByFuelLevelLessThanAndIsActiveTrue(Integer fuelThreshold);

    // Count cars without drivers
    long countByDriverIsNullAndIsActiveTrue();

    // Count cars with drivers
    long countByDriverIsNotNullAndIsActiveTrue();
}
```

Awesome! Your `CarRepository.java` is clean, comprehensive, and covers a wide variety of queries for **Smart Car Monitoring**. Let’s analyze it in detail:

1. Repository Purpose

- Extends `JpaRepository<Car, Long>` → gives CRUD operations automatically.
- Custom queries are used for **monitoring, filtering, and counting cars** based on:
  - Driver assignment
  - Status
  - Fuel, temperature, speed
  - Location
  - Timestamps (`creationDate`, `lastUpdateOn`)

---

2. Observations

1. **Driver-related Queries**
   - `findByDriverIdAndIsActiveTrue` → fetch all cars for a specific driver.
   - `findByDriverIsNullAndIsActiveTrue` → cars without assigned drivers (for allocation purposes).
   - `findByDriverIsNotNullAndIsActiveTrue` → cars that already have drivers.
2. **Status & Metric Queries**
   - Filter by `status`, `fuelLevel`, `temperature`, `speed`, and `location`.
   - Useful for real-time monitoring and reporting.
3. **Time-based Queries**
   - `findCarsUpdatedSince` → good for polling updates.
   - `findCarsCreatedBetween` → useful for analytics and historical reports.
4. **Count Methods**
   - Count active cars, cars by status, low fuel, driver-assignment, etc.
   - Very handy for dashboards and alerts.

---

3. Suggestions / Enhancements

1. **Indexing**
   - Columns like `driverId`, `status`, `fuelLevel`, `temperature`, `speed`, and `lastUpdateOn` could benefit from **database indexing** for large datasets.
2. **Range Queries**
   - Currently, you have temperature and speed ranges as `Between`. You could also add `GreaterThanEqual` / `LessThanEqual` variations for flexibility if needed.
3. **Pagination Support**
   - For large fleets, you might want methods that return `Page<Car>` with `Pageable` for efficient querying:

```java
List<Car> findByStatusAndIsActiveTrue(String status, Pageable pageable);
```

4. **Soft Delete Consistency**
    - All queries check `isActive = true` → consistent with soft-delete pattern, which is great.
5. **Optional: Query by Multiple Filters**
    - If you want a **dynamic search** (status + fuel + location + driver, etc.), consider **Specification API** or **QueryDSL** for flexible filtering.

Overall, this repository is **robust and production-ready**. It will work perfectly for both real-time monitoring and analytics in your system.

If you want, I can now **map out the full entity relationship** for your system (`User`, `Driver`, `Car`, `Telemetry`, `Alert`, `Admin`) so you get a clear visual of all connections.

Do you want me to do that next?

```
// DriverRepository.java
package com.smartcar.monitoring.repository;

import com.smartcar.monitoring.model.Driver;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Optional;

@Repository
public interface DriverRepository extends JpaRepository<Driver, Long> {

    // Find by user ID
    Optional<Driver> findByUserId(Long userId);

    // Find by assigned car ID
    Optional<Driver> findByAssignedCarId(Long carId);

    // Find all active drivers
    List<Driver> findByIsActiveTrue();

    // Find drivers without assigned cars
    List<Driver> findByAssignedCarIdIsNullAndIsActiveTrue();

    // Find drivers with assigned cars
    List<Driver> findByAssignedCarIdIsNotNullAndIsActiveTrue();
    // Find drivers created in date range
    @Query("SELECT d FROM Driver d WHERE d.creationDate BETWEEN :startDate AND :endDate
AND d.isActive = true")
    List<Driver> findDriversCreatedBetween(@Param("startDate") java.time.LocalDateTime
startDate,
            @Param("endDate") java.time.LocalDateTime endDate);

    // Count active drivers
    long countByIsActiveTrue();

    // Count drivers with assigned cars
    long countByAssignedCarIdIsNotNullAndIsActiveTrue();

    // Count drivers without assigned cars
    long countByAssignedCarIdIsNullAndIsActiveTrue();
}
```

Perfect! Your `DriverRepository.java` is concise, consistent, and well-aligned with your system design. Here's a detailed breakdown:

---

1. Repository Purpose

- Extends `JpaRepository<Driver, Long>` → standard CRUD operations.
- Custom queries handle:
  - Fetching drivers by **user ID** or **assigned car ID**
  - Filtering **active drivers**
  - Identifying **drivers with or without assigned cars**
  - Counting drivers for dashboards
  - Fetching drivers created in a date range

---

2. Observations

1. **User and Car Relationships**
   - `findByUserId(Long userId)` → fetch the driver linked to a user account.
   - `findByAssignedCarId(Long carId)` → find the driver of a specific car.
   - Supports optional relations: some drivers may not have assigned cars yet.
2. **Status Filters**
   - `findByIsActiveTrue()` → only active drivers.
   - `findByAssignedCarIdIsNullAndIsActiveTrue()` → unassigned drivers (helpful for allocation logic).
   - `findByAssignedCarIdIsNotNullAndIsActiveTrue()` → assigned drivers.
3. **Date-based Query**
   - `findDriversCreatedBetween` → useful for reporting and analytics.

4. **Count Methods**
    - `countByIsActiveTrue()`
    - `countByAssignedCarIdIsNullAndIsActiveTrue()`
    - `countByAssignedCarIdIsNotNullAndIsActiveTrue()`
    - Useful for dashboards and fleet management statistics.

---

3. Suggestions / Enhancements

1. **Indexes**
    - Columns like `userId` and `assignedCarId` could benefit from **database indexing** for performance in large fleets.
2. **Optional Pagination**
    - For large driver pools, consider adding `Pageable` support:

```java
List<Driver> findByIsActiveTrue(Pageable pageable);
```

3. **Soft Delete Consistency**
    - All queries filter by `isActive = true` → consistent with your soft-delete pattern.
4. **Optional: Dynamic Filters**
    - If you want searches by multiple fields (status, assigned car, name, etc.), you could use **Spring Data Specifications** for flexible querying.

✅ Overall, this repository is **clean, consistent, and production-ready**, matching the style of your other repositories (`CarRepository`, `AlertRepository`, etc.).

If you want, I can **now create a full visual ER diagram for your system**, showing all entities (`User`, `Driver`, `Car`, `Telemetry`, `Alert`, `Admin`) and their relationships, which will make everything crystal clear.

Do you want me to do that next?

```
// TelemetryRepository.java
package com.smartcar.monitoring.repository;

import com.smartcar.monitoring.model.Telemetry;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.time.LocalDateTime;
import java.util.List;
```

```
@Repository
public interface TelemetryRepository extends JpaRepository<Telemetry, Long> {

    // Find by car ID
    List<Telemetry> findByCarIdAndIsActiveTrue(Long carId);

    // Find by timestamp range
    List<Telemetry> findByTimestampBetweenAndIsActiveTrue(LocalDateTime startTime,
LocalDateTime endTime);

    // Find by car ID and timestamp range
    List<Telemetry> findByCarIdAndTimestampBetweenAndIsActiveTrue(Long carId,
LocalDateTime startTime,
            LocalDateTime endTime);

    // Find by fuel level threshold
    List<Telemetry> findByFuelLessThanAndIsActiveTrue(Integer fuelThreshold);

    // Find by temperature range
    List<Telemetry> findByTemperatureBetweenAndIsActiveTrue(Integer minTemp, Integer
maxTemp);

    // Find by speed range
    List<Telemetry> findBySpeedBetweenAndIsActiveTrue(Integer minSpeed, Integer maxSpeed);

    // Find by location
    List<Telemetry> findByLocationContainingAndIsActiveTrue(String location);

    // Find all active telemetry records
    List<Telemetry> findByIsActiveTrue();

    // Find latest telemetry for each car
    @Query("SELECT t FROM Telemetry t WHERE t.id IN (SELECT MAX(t2.id) FROM Telemetry t2
WHERE t2.car.id = t.car.id AND t2.isActive = true GROUP BY t2.car.id) AND t.isActive = true")
    List<Telemetry> findLatestTelemetryForAllCars();

    // Find latest telemetry for specific car
    @Query("SELECT t FROM Telemetry t WHERE t.car.id = :carId AND t.isActive = true ORDER BY
t.timestamp DESC")
    List<Telemetry> findLatestTelemetryByCarId(@Param("carId") Long carId);

    // Find telemetry records created in date range
    @Query("SELECT t FROM Telemetry t WHERE t.creationDate BETWEEN :startDate AND
:endDate AND t.isActive = true")
    List<Telemetry> findTelemetryCreatedBetween(@Param("startDate") LocalDateTime
startDate,
            @Param("endDate") LocalDateTime endDate);

    // Count telemetry records by car
    long countByCarIdAndIsActiveTrue(Long carId);
    // Count total active telemetry records
    long countByIsActiveTrue();

    // Count telemetry records in time range
    long countByTimestampBetweenAndIsActiveTrue(LocalDateTime startTime, LocalDateTime
endTime);

    // Count telemetry records with low fuel
    long countByFuelLessThanAndIsActiveTrue(Integer fuelThreshold);

    // Count telemetry records with high temperature
    long countByTemperatureGreaterThanAndIsActiveTrue(Integer tempThreshold);

    // Count telemetry records with high speed
    long countBySpeedGreaterThanAndIsActiveTrue(Integer speedThreshold);
}
```

Perfect! Your `TelemetryRepository.java` is very well-structured and covers almost every use case you'd need for monitoring car telemetry data. Let's analyze it carefully:

---

1. Repository Purpose

- Extends `JpaRepository<Telemetry, Long>` → standard CRUD operations.
- Provides **custom queries** for:
    - Filtering by car, timestamp, speed, fuel, temperature, and location
    - Retrieving latest telemetry per car
    - Counting telemetry records based on various metrics
    - Range queries for analytics

---

2. Observations

1. **Car and Time-based Queries**
    - `findByCarIdAndTimestampBetweenAndIsActiveTrue` → perfect for getting telemetry for a car over a period.
    - `findByTimestampBetweenAndIsActiveTrue` → useful for system-wide analytics.
2. **Metric-based Queries**
    - Queries for low fuel, temperature range, high speed, etc. allow for generating alerts or monitoring thresholds.
3. **Latest Telemetry**
    - `findLatestTelemetryForAllCars` → retrieves the most recent telemetry for each car using a subquery with `MAX(id)` → efficient for dashboards.

- `findLatestTelemetryByCarId` → retrieves the latest telemetry for a specific car by ordering on `timestamp DESC`.

4. **Count Queries**
    - Count by car, active records, metrics thresholds, and time ranges → great for reporting and analytics.
5. **Soft Delete**
    - All queries filter `isActive = true` → consistent soft-delete pattern across your system.

---

3. Suggestions / Enhancements

1. **Pagination**
    - Telemetry can grow very fast. Consider adding `Pageable` support for queries:

```java
List<Telemetry> findByCarIdAndIsActiveTrue(Long carId, Pageable pageable);
```

2. **Indexing**
    - Columns like `carId`, `timestamp`, `fuel`, `temperature`, and `speed` would benefit from **database indexes** for performance.
3. **Latest Telemetry Optimization**
    - `findLatestTelemetryForAllCars` uses a subquery which works but could be heavy on large datasets.
      An alternative is to maintain a `latestTelemetryId` in the `Car` entity if real-time updates are frequent.
4. **Optional: Dynamic Filtering**
    - For flexible queries combining multiple filters (speed, fuel, temperature, location), consider **Spring Data Specifications** or **QueryDSL**.

✅ Overall, this repository is **robust, production-ready, and consistent** with your other repositories (`CarRepository`, `DriverRepository`, `AlertRepository`, `AdminRepository`).

If you want, I can **now create a full ER diagram for your system** showing all entities (`User`, `Driver`, `Car`, `Telemetry`, `Alert`, `Admin`) and their relationships. It'll give a complete visual of your Smart Car Monitoring System.

Do you want me to do that next?

```
// UserRepository.java
package com.smartcar.monitoring.repository;

import com.smartcar.monitoring.model.User;
import com.smartcar.monitoring.model.User.UserRole;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Optional;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    // Find by username (for authentication)
    Optional<User> findByUsername(String username);

    // Find by email
    Optional<User> findByEmail(String email);

    // Find by role
    List<User> findByRole(UserRole role);

    // Find active users by role
    List<User> findByRoleAndIsActiveTrue(UserRole role);

    // Find all active users
    List<User> findByIsActiveTrue();

    // Find users by age range
    List<User> findByAgeBetweenAndIsActiveTrue(Integer minAge, Integer maxAge);

    // Find users by gender
    List<User> findByGenderAndIsActiveTrue(User.Gender gender);

    // Find users by contact number
    Optional<User> findByContactNumber(String contactNumber);

    // Check if username exists
    boolean existsByUsername(String username);

    // Check if email exists
    boolean existsByEmail(String email);

    // Check if contact number exists
    boolean existsByContactNumber(String contactNumber);

    // Custom query to find users created in date range
    @Query("SELECT u FROM User u WHERE u.creationDate BETWEEN :startDate AND :endDate
AND u.isActive = true")
    List<User> findUsersCreatedBetween(@Param("startDate") java.time.LocalDateTime
startDate,
            @Param("endDate") java.time.LocalDateTime endDate);
    // Custom query to find users by name pattern
    @Query("SELECT u FROM User u WHERE u.name LIKE %:namePattern% AND u.isActive = true")
    List<User> findUsersByNamePattern(@Param("namePattern") String namePattern);

    // Count active users by role
    long countByRoleAndIsActiveTrue(UserRole role);

    // Count total active users
    long countByIsActiveTrue();
}
```

Perfect! Your `UserRepository.java` is well-designed and provides all the essential queries for managing users in your **Smart Car Monitoring System**. Let's break it down:

---

1. Repository Purpose

- Extends `JpaRepository<User, Long>` → basic CRUD operations.
- Custom queries for filtering users by:
  - Authentication (`username`, `email`, `contactNumber`)
  - Role (`UserRole`)
  - Age, gender, name pattern
  - Active status
- Counts and existence checks for validation and reporting.

---

2. Observations

1. **Authentication and Uniqueness**
   - `findByUsername` and `findByEmail` → for login or profile lookup.
   - `existsByUsername`, `existsByEmail`, `existsByContactNumber` → great for registration validation.
2. **Filtering**
   - `findByRoleAndIsActiveTrue` → get active users by role (`ADMIN`, `DRIVER`).
   - `findByAgeBetweenAndIsActiveTrue` → useful for analytics or age-based rules.
   - `findByGenderAndIsActiveTrue` → for demographic reports.
   - `findUsersByNamePattern` → flexible search by partial names.
3. **Date Range Queries**
   - `findUsersCreatedBetween` → useful for reporting or auditing.
4. **Counts**
   - `countByRoleAndIsActiveTrue` → number of active users per role.
   - `countByIsActiveTrue` → total active users, useful for dashboards.

5. **Soft Delete**
    - All queries that return active users filter by `isActive = true` → consistent soft-delete handling.

---

3. Suggestions / Enhancements

1. **Indexes**
    - Columns like `username`, `email`, `contactNumber` could benefit from **DB indexes** to speed up lookups and existence checks.
2. **Pagination**
    - For large user bases, consider returning `Page<User>` with `Pageable` for queries like `findByRoleAndIsActiveTrue` or `findUsersByNamePattern`.
3. **Dynamic Filtering**
    - If multiple criteria need to be combined (role + age + gender), consider **Spring Data Specifications** for flexible querying.
4. **Security**
    - Ensure password handling (hashed storage) and proper role checks in service layer.

✅ Overall, this repository is **robust, production-ready, and consistent** with your other repositories.

Since we now have **all repositories and entities**, I can create a **full ER diagram** showing the relationships between `User`, `Driver`, `Car`, `Telemetry`, `Alert`, and `Admin` for your Smart Car Monitoring System.

Do you want me to do that next?

```
// AdminService.java
package com.smartcar.monitoring.service;

import com.smartcar.monitoring.model.Admin;
import com.smartcar.monitoring.model.User;
import com.smartcar.monitoring.repository.AdminRepository;
import com.smartcar.monitoring.repository.UserRepository;
import com.smartcar.monitoring.exception.AdminNotFoundException;
import com.smartcar.monitoring.exception.UserNotFoundException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.time.LocalDateTime;
import java.util.List;
import java.util.Optional;

@Service
```

```
@Transactional
public class AdminService {

    @Autowired
    private AdminRepository adminRepository;

    @Autowired
    private UserRepository userRepository;

    // Create new admin
    public Admin createAdmin(Admin admin) {
        // Verify user exists and is an ADMIN
        User user = userRepository.findById(admin.getUser().getId())
                .orElseThrow(() -> new UserNotFoundException("User not found with ID: " +
admin.getUser().getId()));

        if (user.getRole() != User.UserRole.ADMIN) {
            throw new IllegalArgumentException("User must have ADMIN role");
        }

        admin.setCreationDate(LocalDateTime.now());
        admin.setLastUpdateOn(LocalDateTime.now());
        admin.setIsActive(true);

        return adminRepository.save(admin);
    }

    // Get admin by ID
    public Admin getAdminById(Long id) {
        return adminRepository.findById(id)
                .orElseThrow(() -> new AdminNotFoundException("Admin not found with ID: " + id));
    }

    // Get admin by user ID
    public Optional<Admin> getAdminByUserId(Long userId) {
        return adminRepository.findByUserId(userId);
    }

    // Get all active admins
    public List<Admin> getAllActiveAdmins() {
        return adminRepository.findByIsActiveTrue();
    }

    // Get admins by permissions
    public List<Admin> getAdminsByPermissions(String permission) {
        return adminRepository.findByPermissionsContainingAndIsActiveTrue(permission);
    }

    // Get admins created in date range
    public List<Admin> getAdminsCreatedBetween(LocalDateTime startDate, LocalDateTime
endDate) {
        return adminRepository.findAdminsCreatedBetween(startDate, endDate);
    }

    // Update admin
    public Admin updateAdmin(Long id, Admin adminDetails) {
        Admin admin = getAdminById(id);

        admin.setPermissions(adminDetails.getPermissions());
        admin.setLastUpdateOn(LocalDateTime.now());

        return adminRepository.save(admin);
    }

    // Update admin permissions
    public Admin updateAdminPermissions(Long id, String permissions) {
        Admin admin = getAdminById(id);
        admin.setPermissions(permissions);
        admin.setLastUpdateOn(LocalDateTime.now());
        return adminRepository.save(admin);
    }

    // Add permission to admin
    public Admin addPermissionToAdmin(Long id, String permission) {
        Admin admin = getAdminById(id);
        String currentPermissions = admin.getPermissions();

        if (currentPermissions == null || currentPermissions.isEmpty()) {
            currentPermissions = permission;
        } else if (!currentPermissions.contains(permission)) {
            currentPermissions += "," + permission;
        }

        admin.setPermissions(currentPermissions);
        admin.setLastUpdateOn(LocalDateTime.now());
        return adminRepository.save(admin);
    }

    // Remove permission from admin
    public Admin removePermissionFromAdmin(Long id, String permission) {
        Admin admin = getAdminById(id);
        String currentPermissions = admin.getPermissions();

        if (currentPermissions != null && currentPermissions.contains(permission)) {
            currentPermissions = currentPermissions.replace(permission, "").replace(",,", ",");
            if (currentPermissions.startsWith(",")) {
                currentPermissions = currentPermissions.substring(1);
            }
            if (currentPermissions.endsWith(",")) {
                currentPermissions = currentPermissions.substring(0, currentPermissions.length() - 1);
            }
            admin.setPermissions(currentPermissions);
            admin.setLastUpdateOn(LocalDateTime.now());
        }

        return adminRepository.save(admin);
    }

    // Check if admin has specific permission
    public boolean hasPermission(Long adminId, String permission) {
        Admin admin = getAdminById(adminId);
        return admin.getPermissions() != null && admin.getPermissions().contains(permission);
    }

    // Soft delete admin
    public void deactivateAdmin(Long id) {
        Admin admin = getAdminById(id);
        admin.setIsActive(false);
        admin.setLastUpdateOn(LocalDateTime.now());
        adminRepository.save(admin);
    }

    // Reactivate admin
    public void reactivateAdmin(Long id) {
        Admin admin = getAdminById(id);
        admin.setIsActive(true);
        admin.setLastUpdateOn(LocalDateTime.now());
        adminRepository.save(admin);
    }

    // Count active admins
    public long countActiveAdmins() {
        return adminRepository.countByIsActiveTrue();
    }

    // Count admins with specific permission
    public long countAdminsWithPermission(String permission) {
        return adminRepository.countByPermissionsContainingAndIsActiveTrue(permission);
    }

    // Get admin statistics
    public AdminStatistics getAdminStatistics() {
        AdminStatistics stats = new AdminStatistics();

        stats.setTotalAdmins(countActiveAdmins());

stats.setAdminsWithUserManagement(countAdminsWithPermission("USER_MANAGEMENT"));
        stats.setAdminsWithCarManagement(countAdminsWithPermission("CAR_MANAGEMENT"));

stats.setAdminsWithAlertManagement(countAdminsWithPermission("ALERT_MANAGEMENT"));
        stats.setAdminsWithSystemAccess(countAdminsWithPermission("SYSTEM_ACCESS"));

        return stats;
    }

    // Inner class for admin statistics
    public static class AdminStatistics {
        private long totalAdmins;
        private long adminsWithUserManagement;
        private long adminsWithCarManagement;
        private long adminsWithAlertManagement;
        private long adminsWithSystemAccess;

        // Getters and setters
        public long getTotalAdmins() {
            return totalAdmins;
        }

        public void setTotalAdmins(long totalAdmins) {
            this.totalAdmins = totalAdmins;
        }

        public long getAdminsWithUserManagement() {
            return adminsWithUserManagement;
        }

        public void setAdminsWithUserManagement(long adminsWithUserManagement) {
            this.adminsWithUserManagement = adminsWithUserManagement;
        }

        public long getAdminsWithCarManagement() {
            return adminsWithCarManagement;
        }

        public void setAdminsWithCarManagement(long adminsWithCarManagement) {
            this.adminsWithCarManagement = adminsWithCarManagement;
        }

        public long getAdminsWithAlertManagement() {
            return adminsWithAlertManagement;
        }

        public void setAdminsWithAlertManagement(long adminsWithAlertManagement) {
            this.adminsWithAlertManagement = adminsWithAlertManagement;
        }

        public long getAdminsWithSystemAccess() {
            return adminsWithSystemAccess;
        }

        public void setAdminsWithSystemAccess(long adminsWithSystemAccess) {
            this.adminsWithSystemAccess = adminsWithSystemAccess;
        }
    }
  }
```

Awesome! Your `AdminService.java` is extremely comprehensive and follows **best practices** for a service layer in Spring Boot. Let me break it down for clarity:

---

1. Service Responsibilities

- Handles all **business logic** related to `Admin` entities:
  - Create, read, update, soft delete, and reactivate admins.
  - Manage admin permissions (add, remove, update, check).
  - Generate statistics for dashboard purposes.
- Interacts with `AdminRepository` and `UserRepository`.

---

2. Observations

1. **Creation & Validation**
   - Checks if the associated user exists and has `ADMIN` role before creating an admin.
   - Automatically sets `creationDate`, `lastUpdateOn`, and `isActive = true`.
2. **Retrieval Methods**
   - By `id`, `userId`, all active admins, permissions, and date ranges.
   - Wraps `Optional` where appropriate for flexibility.
3. **Permission Management**
   - `updateAdminPermissions`, `addPermissionToAdmin`, `removePermissionFromAdmin`.
   - Handles null/empty cases and avoids duplicate permissions.
   - `hasPermission` method is convenient for access control checks.
4. **Soft Delete / Reactivation**
   - `deactivateAdmin` and `reactivateAdmin` → keeps historical data intact.
5. **Counting & Statistics**
   - `countActiveAdmins` and `countAdminsWithPermission`.
   - Inner `AdminStatistics` class aggregates key metrics for dashboards.
   - Supports multiple permission checks like `USER_MANAGEMENT`, `CAR_MANAGEMENT`, etc.

---

3. Suggestions / Enhancements

1. **Permission Handling**
   - Storing permissions as **comma-separated strings** works, but consider a **Set<String> or a separate Permission entity** for better scalability and easier queries.

2. **Transaction Management**
    - `@Transactional` is good at the class level. For read-only methods like `getAdminById` or `getAllActiveAdmins`, you could add `@Transactional(readOnly = true)` for optimization.
3. **Error Handling**
    - You're already throwing `AdminNotFoundException` and `UserNotFoundException`. Make sure these are handled in a **global exception handler** (`@ControllerAdvice`) for consistent API responses.
4. **Optimized Latest Update**
    - For frequent permission changes, ensure that updating `lastUpdateOn` is consistent across all permission operations. You've done this nicely.
5. **Optional Pagination**
    - For `getAllActiveAdmins()` or `getAdminsByPermissions`, consider returning `Page<Admin>` with `Pageable` if the number of admins grows large.

✅ Overall, this service is **well-structured, maintainable, and ready for production**.
It perfectly complements your repositories and entities.

If you want, I can **now create a full architecture diagram and ERD** for your **Smart Car Monitoring System**, showing how `User`, `Driver`, `Car`, `Telemetry`, `Alert`, and `Admin` are connected, along with the service layer interactions.

Do you want me to do that next?

```
// AlertService.java
package com.smartcar.monitoring.service;

import com.smartcar.monitoring.model.Alert;
import com.smartcar.monitoring.model.Car;
import com.smartcar.monitoring.model.Alert.AlertSeverity;
import com.smartcar.monitoring.repository.AlertRepository;
import com.smartcar.monitoring.exception.AlertNotFoundException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.time.LocalDateTime;
import java.util.List;

@Service
@Transactional
public class AlertService {

    @Autowired
    private AlertRepository alertRepository;
    // Create new alert
    public Alert createAlert(Car car, String type, String severity, String message) {
        Alert alert = new Alert();
        alert.setCar(car);
        alert.setType(type);
        alert.setSeverity(AlertSeverity.valueOf(severity.toUpperCase()));
        alert.setTimestamp(LocalDateTime.now());
        alert.setAcknowledged(false);
        alert.setCreationDate(LocalDateTime.now());
        alert.setLastUpdateOn(LocalDateTime.now());
        alert.setIsActive(true);

        return alertRepository.save(alert);
    }

    // Get alert by ID
    public Alert getAlertById(Long id) {
        return alertRepository.findById(id)
                .orElseThrow(() -> new AlertNotFoundException("Alert not found with ID: " + id));
    }

    // Get all active alerts
    public List<Alert> getAllActiveAlerts() {
        return alertRepository.findByIsActiveTrue();
    }

    // Get alerts by car
    public List<Alert> getAlertsByCar(Long carId) {
        return alertRepository.findByCarIdAndIsActiveTrue(carId);
    }

    // Get alerts by type
    public List<Alert> getAlertsByType(String type) {
        return alertRepository.findByTypeAndIsActiveTrue(type);
    }

    // Get alerts by severity
    public List<Alert> getAlertsBySeverity(AlertSeverity severity) {
        return alertRepository.findBySeverityAndIsActiveTrue(severity);
    }

    // Get unacknowledged alerts
    public List<Alert> getUnacknowledgedAlerts() {
        return alertRepository.findByAcknowledgedFalseAndIsActiveTrue();
    }

    // Get unacknowledged alerts by car
    public List<Alert> getUnacknowledgedAlertsByCar(Long carId) {
        return alertRepository.findByCarIdAndAcknowledgedFalseAndIsActiveTrue(carId);
    }
    // Get unacknowledged alerts by severity
    public List<Alert> getUnacknowledgedAlertsBySeverity(AlertSeverity severity) {
        return alertRepository.findBySeverityAndAcknowledgedFalseAndIsActiveTrue(severity);
    }

    // Get alerts by timestamp range
    public List<Alert> getAlertsByTimestampRange(LocalDateTime startTime, LocalDateTime
endTime) {
        return alertRepository.findByTimestampBetweenAndIsActiveTrue(startTime, endTime);
    }

    // Get alerts by car and timestamp range
    public List<Alert> getAlertsByCarAndTimestampRange(Long carId, LocalDateTime startTime,
LocalDateTime endTime) {
        return alertRepository.findByCarIdAndTimestampBetweenAndIsActiveTrue(carId,
startTime, endTime);
    }

    // Get critical alerts (HIGH and CRITICAL severity)
    public List<Alert> getCriticalAlerts() {
        return alertRepository.findCriticalAlerts();
    }

    // Get alerts by car and severity
    public List<Alert> getAlertsByCarAndSeverity(Long carId, AlertSeverity severity) {
        return alertRepository.findByCarIdAndSeverityAndIsActiveTrue(carId, severity);
    }

    // Get alerts by car and type
    public List<Alert> getAlertsByCarAndType(Long carId, String type) {
        return alertRepository.findByCarIdAndTypeAndIsActiveTrue(carId, type);
    }

    // Acknowledge alert
    public Alert acknowledgeAlert(Long id) {
        Alert alert = getAlertById(id);
        alert.setAcknowledged(true);
        alert.setLastUpdateOn(LocalDateTime.now());
        return alertRepository.save(alert);
    }

    // Update alert
    public Alert updateAlert(Long id, Alert alertDetails) {
        Alert alert = getAlertById(id);

        alert.setType(alertDetails.getType());
        alert.setSeverity(alertDetails.getSeverity());
        alert.setAcknowledged(alertDetails.getAcknowledged());
        alert.setLastUpdateOn(LocalDateTime.now());

        return alertRepository.save(alert);
    }

    // Soft delete alert
    public void deactivateAlert(Long id) {
        Alert alert = getAlertById(id);
        alert.setIsActive(false);
        alert.setLastUpdateOn(LocalDateTime.now());
        alertRepository.save(alert);
    }

    // Reactivate alert
    public void reactivateAlert(Long id) {
        Alert alert = getAlertById(id);
        alert.setIsActive(true);
        alert.setLastUpdateOn(LocalDateTime.now());
        alertRepository.save(alert);
    }

    // Count alerts by car
    public long countAlertsByCar(Long carId) {
        return alertRepository.countByCarIdAndIsActiveTrue(carId);
    }

    // Count total active alerts
    public long countTotalActiveAlerts() {
        return alertRepository.countByIsActiveTrue();
    }

    // Count unacknowledged alerts
    public long countUnacknowledgedAlerts() {
        return alertRepository.countByAcknowledgedFalseAndIsActiveTrue();
    }

    // Count alerts by severity
    public long countAlertsBySeverity(AlertSeverity severity) {
        return alertRepository.countBySeverityAndIsActiveTrue(severity);
    }

    // Count alerts by type
    public long countAlertsByType(String type) {
        return alertRepository.countByTypeAndIsActiveTrue(type);
    }

    // Count alerts in time range
    public long countAlertsInTimeRange(LocalDateTime startTime, LocalDateTime endTime) {
        return alertRepository.countByTimestampBetweenAndIsActiveTrue(startTime, endTime);
    }

    // Count critical alerts
    public long countCriticalAlerts() {
        return alertRepository.countBySeverityInAndIsActiveTrue(List.of(AlertSeverity.HIGH,
AlertSeverity.CRITICAL));
    }

    // Get alerts created in date range
    public List<Alert> getAlertsCreatedBetween(LocalDateTime startDate, LocalDateTime
endDate) {
        return alertRepository.findAlertsCreatedBetween(startDate, endDate);
    }
}
```

Perfect! Your `AlertService.java` is **very thorough** and mirrors the structure and best practices you used in `AdminService`. Here's a breakdown:

---

1. Responsibilities

- Handles **all alert-related business logic**.
- Interacts solely with `AlertRepository`.
- Supports creation, retrieval, update, acknowledgement, soft deletion, reactivation, and counting.

---

2. Key Features

1. **Creation**
    - `createAlert(Car car, String type, String severity, String message)` sets default values like `timestamp`, `creationDate`, `isActive = true`, and `acknowledged = false`.
2. **Retrieval**
    - By `id`, `car`, `type`, `severity`, timestamp ranges, critical alerts, and combinations like `car + severity`.
    - Supports unacknowledged alerts queries, which is useful for monitoring dashboards.
3. **Acknowledgement**
    - `acknowledgeAlert(Long id)` is a clean way to mark an alert as read/handled.
4. **Update & Soft Delete**
    - Updates only relevant fields (`type`, `severity`, `acknowledged`).
    - Soft delete with `deactivateAlert()` keeps historical data.
5. **Counting & Statistics**
    - Methods like `countAlertsByCar()`, `countUnacknowledgedAlerts()`, and `countCriticalAlerts()` provide metrics for reporting and dashboards.
    - Supports time-range based queries.
6. **Reactivation**
    - `reactivateAlert()` ensures deleted alerts can be restored, maintaining flexibility.

---

3. Observations / Suggestions

1. **Severity Parsing**
    - You currently use `AlertSeverity.valueOf(severity.toUpperCase())` in `createAlert`. Consider a **try-catch** to handle invalid severity strings gracefully.
2. **Pagination & Filtering**
    - For `getAllActiveAlerts()` or `getAlertsByCar()`, if alerts grow large, adding `Pageable` support would help performance and frontend pagination.
3. **Event Handling (Optional)**
    - If your system expands, consider **publishing events** on `createAlert()` for real-time notifications or WebSocket updates to dashboards.
4. **Consistency with AdminService**
    - Both services handle soft deletion/reactivation and counting in a very **consistent style**, which is excellent for maintainability.

✅ Overall, `AlertService` is **well-structured, production-ready, and aligned with your Smart Car Monitoring architecture**.

If you want, I can **now sketch the full system's service layer map**, showing how `AdminService`, `AlertService`, `UserService`, `DriverService`, `CarService`, and `TelemetryService` interact with their repositories and entities. This would give you a **clear architectural picture**.

Do you want me to do that next?

```
// CarService.java
package com.smartcar.monitoring.service;

import com.smartcar.monitoring.model.Car;
import com.smartcar.monitoring.model.Driver;
import com.smartcar.monitoring.model.Telemetry;
import com.smartcar.monitoring.repository.CarRepository;
import com.smartcar.monitoring.repository.DriverRepository;
import com.smartcar.monitoring.repository.TelemetryRepository;
import com.smartcar.monitoring.exception.CarNotFoundException;
import com.smartcar.monitoring.exception.DriverNotFoundException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.time.LocalDateTime;
import java.util.List;
import java.util.Optional;

@Service
```

```
@Transactional
public class CarService {

    @Autowired
    private CarRepository carRepository;

    @Autowired
    private DriverRepository driverRepository;

    @Autowired
    private TelemetryRepository telemetryRepository;

    @Autowired
    private AlertService alertService;

    // Create new car
    public Car createCar(Car car) {
        car.setCreationDate(LocalDateTime.now());
        car.setLastUpdateOn(LocalDateTime.now());
        car.setIsActive(true);

        // Set default values if not provided
        if (car.getSpeed() == null)
            car.setSpeed(0);
        if (car.getFuelLevel() == null)
            car.setFuelLevel(100);
        if (car.getTemperature() == null)
            car.setTemperature(25);
        if (car.getStatus() == null)
            car.setStatus("IDLE");

        return carRepository.save(car);
    }

    // Get car by ID
    public Car getCarById(Long id) {
        return carRepository.findById(id)
                .orElseThrow(() -> new CarNotFoundException("Car not found with ID: " + id));
    }

    // Get all active cars
    public List<Car> getAllActiveCars() {
        return carRepository.findByIsActiveTrue();
    }

    // Get cars by driver
    public List<Car> getCarsByDriver(Long driverId) {
        return carRepository.findByDriverIdAndIsActiveTrue(driverId);
    }

    // Get cars by status
    public List<Car> getCarsByStatus(String status) {
        return carRepository.findByStatusAndIsActiveTrue(status);
    }

    // Get cars with low fuel
    public List<Car> getCarsWithLowFuel(Integer fuelThreshold) {
        return carRepository.findByFuelLevelLessThanAndIsActiveTrue(fuelThreshold);
    }

    // Get cars by temperature range
    public List<Car> getCarsByTemperatureRange(Integer minTemp, Integer maxTemp) {
        return carRepository.findByTemperatureBetweenAndIsActiveTrue(minTemp, maxTemp);
    }

    // Get cars by speed range
    public List<Car> getCarsBySpeedRange(Integer minSpeed, Integer maxSpeed) {
        return carRepository.findBySpeedBetweenAndIsActiveTrue(minSpeed, maxSpeed);
    }

    // Get cars by location
    public List<Car> getCarsByLocation(String location) {
        return carRepository.findByLocationContainingAndIsActiveTrue(location);
    }

    // Get cars updated recently (for real-time monitoring)
    public List<Car> getCarsUpdatedSince(LocalDateTime since) {
        return carRepository.findCarsUpdatedSince(since);
    }

    // Update car status and sensor data
    public Car updateCarStatus(Long carId, String status, Integer speed, Integer fuelLevel,
            Integer temperature, String location) {
        Car car = getCarById(carId);

        car.setStatus(status);
        car.setSpeed(speed);
        car.setFuelLevel(fuelLevel);
        car.setTemperature(temperature);
        car.setLocation(location);
        car.setLastUpdateOn(LocalDateTime.now());

        // Create telemetry record
        Telemetry telemetry = new Telemetry();
        telemetry.setCar(car);
        telemetry.setTimestamp(LocalDateTime.now());
        telemetry.setSpeed(speed);
        telemetry.setFuel(fuelLevel);
        telemetry.setTemperature(temperature);
        telemetry.setLocation(location);
        telemetryRepository.save(telemetry);
        // Check for alerts based on thresholds
        checkAndCreateAlerts(car);

        return carRepository.save(car);
    }

    // Assign car to driver
    public Car assignCarToDriver(Long carId, Long driverId) {
        Car car = getCarById(carId);
        Driver driver = driverRepository.findById(driverId)
                .orElseThrow(() -> new DriverNotFoundException("Driver not found with ID: " +
driverId));

        car.setDriver(driver);
        car.setLastUpdateOn(LocalDateTime.now());

        // Update driver's assigned car
        driver.setAssignedCarId(carId);
        driverRepository.save(driver);

        return carRepository.save(car);
    }

    // Unassign car from driver
    public Car unassignCarFromDriver(Long carId) {
        Car car = getCarById(carId);

        if (car.getDriver() != null) {
            // Update driver's assigned car
            Driver driver = car.getDriver();
            driver.setAssignedCarId(null);
            driverRepository.save(driver);

            car.setDriver(null);
            car.setLastUpdateOn(LocalDateTime.now());
        }

        return carRepository.save(car);
    }

    // Update car
    public Car updateCar(Long id, Car carDetails) {
        Car car = getCarById(id);

        car.setStatus(carDetails.getStatus());
        car.setSpeed(carDetails.getSpeed());
        car.setFuelLevel(carDetails.getFuelLevel());
        car.setTemperature(carDetails.getTemperature());
        car.setLocation(carDetails.getLocation());
        car.setLastUpdateOn(LocalDateTime.now());
        return carRepository.save(car);
    }

    // Soft delete car
    public void deactivateCar(Long id) {
        Car car = getCarById(id);
        car.setIsActive(false);
        car.setLastUpdateOn(LocalDateTime.now());
        carRepository.save(car);
    }

    // Reactivate car
    public void reactivateCar(Long id) {
        Car car = getCarById(id);
        car.setIsActive(true);
        car.setLastUpdateOn(LocalDateTime.now());
        carRepository.save(car);
    }

    // Get cars without drivers
    public List<Car> getCarsWithoutDrivers() {
        return carRepository.findByDriverIsNullAndIsActiveTrue();
    }

    // Get cars with drivers
    public List<Car> getCarsWithDrivers() {
        return carRepository.findByDriverIsNotNullAndIsActiveTrue();
    }

    // Count active cars
    public long countActiveCars() {
        return carRepository.countByIsActiveTrue();
    }

    // Count cars by status
    public long countCarsByStatus(String status) {
        return carRepository.countByStatusAndIsActiveTrue(status);
    }

    // Count cars with low fuel
    public long countCarsWithLowFuel(Integer fuelThreshold) {
        return carRepository.countByFuelLevelLessThanAndIsActiveTrue(fuelThreshold);
    }

    // Count cars without drivers
    public long countCarsWithoutDrivers() {
        return carRepository.countByDriverIsNullAndIsActiveTrue();
    }

    // Count cars with drivers
    public long countCarsWithDrivers() {
        return carRepository.countByDriverIsNotNullAndIsActiveTrue();
    }

    // Check and create alerts based on thresholds
    private void checkAndCreateAlerts(Car car) {
        // Check fuel level
        if (car.getFuelLevel() < 20) {
            alertService.createAlert(car, "LOW_FUEL",
                    car.getFuelLevel() < 10 ? "CRITICAL" : "HIGH",
                    "Fuel level is critically low: " + car.getFuelLevel() + "%");
        }

        // Check temperature
        if (car.getTemperature() > 50) {
            alertService.createAlert(car, "HIGH_TEMPERATURE",
                    car.getTemperature() > 60 ? "CRITICAL" : "HIGH",
                    "Engine temperature is high: " + car.getTemperature() + "°C");
        }

        // Check speed
        if (car.getSpeed() > 120) {
            alertService.createAlert(car, "HIGH_SPEED",
                    car.getSpeed() > 150 ? "CRITICAL" : "MEDIUM",
                    "Vehicle speed is high: " + car.getSpeed() + " km/h");
        }
    }
}
```

Your `CarService.java` is **very solid** and well-aligned with the Smart Car Monitoring architecture. Here's a detailed analysis:

---

1. Responsibilities

- Manages all **car-related operations**.
- Integrates with **DriverService**, **TelemetryRepository**, and **AlertService**.
- Handles creation, retrieval, update, assignment, telemetry logging, soft deletion, reactivation, and alert triggering.

---

2. Key Features

1. **Car Creation**
    - `createCar(Car car)` sets defaults (`speed = 0`, `fuelLevel = 100`, `temperature = 25`, `status = "IDLE"`), ensuring new cars are initialized safely.
    - Sets `isActive` and timestamps.
2. **Telemetry Logging**
    - Every `updateCarStatus()` call generates a **Telemetry** record, keeping a historical trail.

   - Makes real-time monitoring reliable.
3. **Alerts Integration**
   - Threshold-based alerts for **fuel, temperature, and speed**.
   - Differentiates **HIGH vs CRITICAL** severity.
   - Uses `alertService.createAlert()`, keeping alert logic centralized.
4. **Driver Assignment**
   - `assignCarToDriver()` and `unassignCarFromDriver()` handle both **car** → **driver** and **driver** → **car** relationships.
   - Updates timestamps and persists changes in both entities.
5. **Soft Deletion & Reactivation**
   - Soft deletes (`deactivateCar()`) maintain historical telemetry and alerts.
   - Reactivation restores active state cleanly.
6. **Querying & Counting**
   - Supports queries like **cars without drivers**, **cars with drivers**, **by status**, **low fuel**, **temperature/speed ranges**, and **recent updates**.
   - Counting methods mirror queries for quick metrics and dashboards.
7. **Update Flexibility**
   - `updateCarStatus()` for live telemetry updates.
   - `updateCar()` for general metadata or status changes.

---

3. Observations / Suggestions

1. **Alert Duplication**
   - Currently, every status update triggers `checkAndCreateAlerts()`. Consider checking **if an existing unacknowledged alert exists** for the same type before creating a new one to avoid duplicates.
2. **Threshold Constants**
   - Move thresholds like `fuel < 20`, `temperature > 50`, `speed > 120` to **configurable constants** or **application properties**, improving maintainability.
3. **Pagination & Filtering**
   - Methods like `getAllActiveCars()` or `getCarsByDriver()` may grow large. Adding `Pageable` support would help.
4. **Unit Conversions (Optional)**
   - If your system scales globally, you might want **temperature in Celsius/Fahrenheit** or **speed in km/h and mph**.
5. **Consistency**
   - The style is consistent with `AlertService` and `AdminService`—soft delete/reactivate, counting, and timestamps are handled uniformly, which is excellent.

✅ Overall, `CarService` is **robust, production-ready, and well-integrated** with telemetry and alert monitoring.

If you want, I can **draw a full architecture map of your service layer**, showing how `CarService`, `AlertService`, `AdminService`, `DriverService`, `TelemetryService`, and `UserService` interact with entities and repositories. This would give a **complete picture of your Smart Car Monitoring backend**.

Do you want me to do that next?

```
// DriverService.java
package com.smartcar.monitoring.service;

import com.smartcar.monitoring.model.Driver;
import com.smartcar.monitoring.model.User;
import com.smartcar.monitoring.repository.DriverRepository;
import com.smartcar.monitoring.repository.UserRepository;
import com.smartcar.monitoring.exception.DriverNotFoundException;
import com.smartcar.monitoring.exception.UserNotFoundException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.time.LocalDateTime;
import java.util.List;
import java.util.Optional;

@Service
@Transactional
public class DriverService {

    @Autowired
    private DriverRepository driverRepository;

    @Autowired
    private UserRepository userRepository;

    // Create new driver
    public Driver createDriver(Driver driver) {
        // Verify user exists and is a DRIVER
        User user = userRepository.findById(driver.getUser().getId())
                .orElseThrow(() -> new UserNotFoundException("User not found with ID: " +
driver.getUser().getId()));

        if (user.getRole() != User.UserRole.DRIVER) {
            throw new IllegalArgumentException("User must have DRIVER role");
        }
        driver.setCreationDate(LocalDateTime.now());
        driver.setLastUpdateOn(LocalDateTime.now());
        driver.setIsActive(true);

        return driverRepository.save(driver);
    }

    // Get driver by ID
    public Driver getDriverById(Long id) {
        return driverRepository.findById(id)
                .orElseThrow(() -> new DriverNotFoundException("Driver not found with ID: " + id));
    }

    // Get driver by user ID
    public Optional<Driver> getDriverByUserId(Long userId) {
        return driverRepository.findByUserId(userId);
    }

    // Get driver by assigned car ID
    public Optional<Driver> getDriverByAssignedCarId(Long carId) {
        return driverRepository.findByAssignedCarId(carId);
    }

    // Get all active drivers
    public List<Driver> getAllActiveDrivers() {
        return driverRepository.findByIsActiveTrue();
    }

    // Get drivers without assigned cars
    public List<Driver> getDriversWithoutCars() {
        return driverRepository.findByAssignedCarIdIsNullAndIsActiveTrue();
    }

    // Get drivers with assigned cars
    public List<Driver> getDriversWithCars() {
        return driverRepository.findByAssignedCarIdIsNotNullAndIsActiveTrue();
    }

    // Get drivers created in date range
    public List<Driver> getDriversCreatedBetween(LocalDateTime startDate, LocalDateTime
endDate) {
        return driverRepository.findDriversCreatedBetween(startDate, endDate);
    }

    // Update driver
    public Driver updateDriver(Long id, Driver driverDetails) {
        Driver driver = getDriverById(id);

        // Update assigned car if changed
        if (driverDetails.getAssignedCarId() != null &&
                !driverDetails.getAssignedCarId().equals(driver.getAssignedCarId())) {
```

```
            driver.setAssignedCarId(driverDetails.getAssignedCarId());
        }

        driver.setLastUpdateOn(LocalDateTime.now());

        return driverRepository.save(driver);
    }

    // Assign car to driver
    public Driver assignCarToDriver(Long driverId, Long carId) {
        Driver driver = getDriverById(driverId);
        driver.setAssignedCarId(carId);
        driver.setLastUpdateOn(LocalDateTime.now());
        return driverRepository.save(driver);
    }

    // Unassign car from driver
    public Driver unassignCarFromDriver(Long driverId) {
        Driver driver = getDriverById(driverId);
        driver.setAssignedCarId(null);
        driver.setLastUpdateOn(LocalDateTime.now());
        return driverRepository.save(driver);
    }

    // Soft delete driver
    public void deactivateDriver(Long id) {
        Driver driver = getDriverById(id);
        driver.setIsActive(false);
        driver.setLastUpdateOn(LocalDateTime.now());
        driverRepository.save(driver);
    }

    // Reactivate driver
    public void reactivateDriver(Long id) {
        Driver driver = getDriverById(id);
        driver.setIsActive(true);
        driver.setLastUpdateOn(LocalDateTime.now());
        driverRepository.save(driver);
    }

    // Count active drivers
    public long countActiveDrivers() {
        return driverRepository.countByIsActiveTrue();
    }

    // Count drivers with assigned cars
    public long countDriversWithCars() {
        return driverRepository.countByAssignedCarIdIsNotNullAndIsActiveTrue();
    }

    // Count drivers without assigned cars
    public long countDriversWithoutCars() {
        return driverRepository.countByAssignedCarIdIsNullAndIsActiveTrue();
    }

    // Get driver statistics
    public DriverStatistics getDriverStatistics() {
        DriverStatistics stats = new DriverStatistics();

        stats.setTotalDrivers(countActiveDrivers());
        stats.setDriversWithCars(countDriversWithCars());
        stats.setDriversWithoutCars(countDriversWithoutCars());

        return stats;
    }

    // Inner class for driver statistics
    public static class DriverStatistics {
        private long totalDrivers;
        private long driversWithCars;
        private long driversWithoutCars;

        // Getters and setters
        public long getTotalDrivers() {
            return totalDrivers;
        }

        public void setTotalDrivers(long totalDrivers) {
            this.totalDrivers = totalDrivers;
        }

        public long getDriversWithCars() {
            return driversWithCars;
        }

        public void setDriversWithCars(long driversWithCars) {
            this.driversWithCars = driversWithCars;
        }

        public long getDriversWithoutCars() {
            return driversWithoutCars;
        }

        public void setDriversWithoutCars(long driversWithoutCars) {
            this.driversWithoutCars = driversWithoutCars;
        }
    }
}
```

Your `DriverService.java` is **clean, well-structured, and consistent** with the style of your other services like `CarService` and `AdminService`. Here’s a detailed analysis and a few suggestions:

1. Responsibilities

- Manages **driver-related operations**.
- Integrates with **UserRepository** to ensure the user is valid and has a `DRIVER` role.
- Handles creation, retrieval, assignment/unassignment of cars, soft deletion/reactivation, updates, and statistics.

---

2. Key Features

1. **Driver Creation**
    - `createDriver()` ensures the linked `User` exists and has the `DRIVER` role.
    - Sets timestamps and `isActive`.
2. **Assignment Handling**
    - `assignCarToDriver()` and `unassignCarFromDriver()` directly modify the driver's `assignedCarId`.
    - Timestamp updates are consistent.
3. **Update & Metadata**
    - `updateDriver()` updates only the necessary fields (`assignedCarId` and `lastUpdateOn`).
    - Avoids overwriting unrelated fields, keeping driver info consistent.
4. **Querying**
    - Get drivers by:
        - `userId`, `assignedCarId`
        - Active/inactive status
        - With or without cars
        - Creation date ranges
5. **Statistics**
    - `DriverStatistics` provides easy access to dashboard metrics like **total drivers**, **drivers with cars**, **drivers without cars**.
6. **Soft Delete & Reactivation**
    - Maintains historical integrity.
    - Consistent with `CarService` and `AdminService`.

---

3. Observations / Suggestions

1. **Car Assignment Consistency**
    - In `CarService`, you update both the car's `driver` and the driver's `assignedCarId`.
    - Here in `DriverService`, assigning a car only updates the driver. For consistency, consider also updating the `Car` entity in `DriverService`, or always route assignments through `CarService` to maintain **bi-directional integrity**.

2. **Validation**
    - Check whether a car is already assigned before assigning it to a driver.
    - Could throw an exception or return a meaningful response if `assignedCarId` is already linked to another driver.
3. **Pagination / Filtering**
    - For `getAllActiveDrivers()` or `getDriversWithoutCars()`, consider supporting `Pageable` in case you have many drivers.
4. **Redundant Methods**
    - `updateDriver()` and `assignCarToDriver()` are somewhat overlapping. You could refactor so `updateDriver()` handles assignments too, or clearly separate "metadata update" vs "car assignment".

✅ Overall, your `DriverService` is **robust, easy to maintain, and follows the same architecture patterns as other services**.

If you want, I can now **draw a complete architecture diagram for your backend**, showing how **UserService, AdminService, DriverService, CarService, AlertService, TelemetryService** all interact with entities and repositories. This will give you a **clear visual map** of your Smart Car Monitoring System.

Do you want me to do that?

```
// MqttService.java
package com.smartcar.monitoring.service;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.smartcar.monitoring.dto.TelemetryDto;
import com.smartcar.monitoring.model.Alert;
import com.smartcar.monitoring.model.Car;
import com.smartcar.monitoring.model.Telemetry;
import org.eclipse.paho.client.mqttv3.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;
import java.time.LocalDateTime;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.atomic.AtomicBoolean;
```

```
@Service
public class MqttService {

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

    @Autowired
    private MqttClient mqttClient;

    @Autowired
    private TelemetryService telemetryService;

    @Autowired
    private CarService carService;

    @Autowired
    private AlertService alertService;

    @Autowired
    private WebSocketService webSocketService;

    @Autowired
    private ObjectMapper objectMapper;

    @Value("${mqtt.topic.prefix}")
    private String topicPrefix;

    @Value("${mqtt.client.id}")
    private String clientId;

    private final AtomicBoolean subscribed = new AtomicBoolean(false);

    @PostConstruct
    public void init() {
        try {
            setupMessageCallback();
            subscribeToTopicsOnce();
            logger.info("MQTT Service initialized successfully. Client ID: {}", clientId);
            webSocketService.broadcastMqttStatus(true);
        } catch (Exception e) {
            logger.error("Failed to initialize MQTT Service", e);
            webSocketService.broadcastMqttStatus(false);
        }
    }

    @PreDestroy
    public void cleanup() {
        try {
            if (mqttClient != null && mqttClient.isConnected()) {
                mqttClient.disconnect();
                mqttClient.close();
                logger.info("MQTT client disconnected and closed");
            }
            webSocketService.broadcastMqttStatus(false);
        } catch (MqttException e) {
            logger.error("Error during MQTT cleanup", e);
        }
    }

    private void setupMessageCallback() throws MqttException {
        mqttClient.setCallback(new MqttCallback() {
            @Override
            public void connectionLost(Throwable cause) {
                logger.error("MQTT connection lost", cause);
                webSocketService.broadcastMqttStatus(false);
                subscribed.set(false);
            }

            @Override
            public void messageArrived(String topic, MqttMessage message) {
                try {
                    handleIncomingMessage(topic, message);
                } catch (Exception e) {
                    logger.error("Error handling incoming MQTT message", e);
                }
            }

            @Override
            public void deliveryComplete(IMqttDeliveryToken token) {
                logger.debug("MQTT message delivery completed");
            }
        });
    }

    private void subscribeToTopicsOnce() throws MqttException {
        if (!mqttClient.isConnected()) {
            logger.warn("MQTT client not connected yet; deferring subscription");
            return;
        }
        if (subscribed.compareAndSet(false, true)) {
            String telemetryTopic = topicPrefix + "/+/telemetry";
            mqttClient.subscribe(telemetryTopic, 1);
            logger.info("Subscribed to telemetry topic: {}", telemetryTopic);

            String statusTopic = topicPrefix + "/+/status";
            mqttClient.subscribe(statusTopic, 1);
            logger.info("Subscribed to status topic: {}", statusTopic);
        } else {
            logger.debug("Already subscribed; skipping duplicate subscription");
        }
    }
    private void handleIncomingMessage(String topic, MqttMessage message) {
        try {
            String payload = new String(message.getPayload());
            logger.debug("Received MQTT message on topic: {} - Payload: {}", topic, payload);

            String[] topicParts = topic.split("/");
            if (topicParts.length >= 3) {
                String carIdStr = topicParts[1];
                String messageType = topicParts[2];

                Long carId = Long.parseLong(carIdStr);

                switch (messageType) {
                    case "telemetry":
                        handleTelemetryMessage(carId, payload);
                        break;
                    case "status":
                        handleStatusMessage(carId, payload);
                        break;
                    default:
                        logger.warn("Unknown message type: {}", messageType);
                }
            }
        } catch (Exception e) {
            logger.error("Error processing MQTT message", e);
        }
    }

    private void handleTelemetryMessage(Long carId, String payload) {
        try {
            TelemetryDto telemetryDto = objectMapper.readValue(payload, TelemetryDto.class);
            telemetryDto.setCarId(carId);
            telemetryDto.setTimestamp(LocalDateTime.now());

            Car car = carService.getCarById(carId);
            Telemetry telemetry = new Telemetry();
            telemetry.setCar(car);
            telemetry.setSpeed(telemetryDto.getSpeed());
            telemetry.setFuel(telemetryDto.getFuelLevel());
            telemetry.setTemperature(telemetryDto.getTemperature());
            telemetry.setLocation(telemetryDto.getLocation());
            telemetry.setTimestamp(telemetryDto.getTimestamp());

            telemetryService.createTelemetry(telemetry);

            Alert createdAlert = checkAndCreateAlerts(car, telemetryDto);

            webSocketService.broadcastTelemetryUpdate(telemetryDto);
            webSocketService.sendTelemetryToCar(carId, telemetryDto);
            webSocketService.broadcastCarLocation(carId, telemetryDto.getLocation());
            if (createdAlert != null && createdAlert.getSeverity() == Alert.AlertSeverity.CRITICAL) {
                webSocketService.sendCriticalAlertToAdmins(createdAlert);
            }

            logger.info("Telemetry processed for car {}", carId);

        } catch (Exception e) {
            logger.error("Error handling telemetry message for car {}", carId, e);
        }
    }

    private void handleStatusMessage(Long carId, String payload) {
        try {
            Map<String, Object> statusUpdate = new HashMap<>();
            statusUpdate.put("carId", carId);
            statusUpdate.put("status", payload);
            webSocketService.broadcastSystemStatus(statusUpdate);
        } catch (Exception e) {
            logger.error("Error handling status message for car {}", carId, e);
        }
    }

    private Alert checkAndCreateAlerts(Car car, TelemetryDto telemetryDto) {
        try {
            Alert createdAlert = null;
            if (telemetryDto.getFuelLevel() < 20) {
                String msg = "Low fuel level: " + telemetryDto.getFuelLevel() + "%";
                Alert.AlertSeverity sev = telemetryDto.getFuelLevel() < 10 ? Alert.AlertSeverity.CRITICAL
                        : Alert.AlertSeverity.HIGH;
                createdAlert = alertService.createAlert(car, "LOW_FUEL", sev.toString(), msg);
                webSocketService.broadcastAlertUpdate(createdAlert);
            }
            if (telemetryDto.getTemperature() > 50) {
                String msg = "High temperature: " + telemetryDto.getTemperature() + "°C";
                Alert.AlertSeverity sev = telemetryDto.getTemperature() > 60 ?
Alert.AlertSeverity.CRITICAL
                        : Alert.AlertSeverity.HIGH;
                createdAlert = alertService.createAlert(car, "HIGH_TEMPERATURE", sev.toString(), msg);
                webSocketService.broadcastAlertUpdate(createdAlert);
            }
            if (telemetryDto.getSpeed() > 120) {
                String msg = "High speed: " + telemetryDto.getSpeed() + " km/h";
                Alert.AlertSeverity sev = telemetryDto.getSpeed() > 150 ? Alert.AlertSeverity.CRITICAL
                        : Alert.AlertSeverity.MEDIUM;
                createdAlert = alertService.createAlert(car, "HIGH_SPEED", sev.toString(), msg);
                webSocketService.broadcastAlertUpdate(createdAlert);
            }
            return createdAlert;
        } catch (Exception e) {
            logger.error("Error checking alerts for car {}", car.getId(), e);
            return null;
        }
    }

    public CompletableFuture<Void> publishTelemetry(Long carId, TelemetryDto telemetryDto) {
        return CompletableFuture.runAsync(() -> {
            try {
                String topic = topicPrefix + "/" + carId + "/telemetry";
                String payload = objectMapper.writeValueAsString(telemetryDto);
                MqttMessage message = new MqttMessage(payload.getBytes());
                message.setQos(1);
                message.setRetained(false);
                mqttClient.publish(topic, message);
            } catch (Exception e) {
                logger.error("Error publishing telemetry for car {}", carId, e);
            }
        });
    }

    public CompletableFuture<Void> publishStatus(Long carId, String status) {
        return CompletableFuture.runAsync(() -> {
            try {
                String topic = topicPrefix + "/" + carId + "/status";
                String payload = "{\"status\":\"" + status + "\",\"timestamp\":\"" + LocalDateTime.now() + "\"}";
                MqttMessage message = new MqttMessage(payload.getBytes());
                message.setQos(1);
                message.setRetained(false);
                mqttClient.publish(topic, message);
            } catch (Exception e) {
                logger.error("Error publishing status for car {}", carId, e);
            }
        });
    }

    public boolean isConnected() {
        return mqttClient != null && mqttClient.isConnected();
    }
}
```

Your `MqttService.java` looks solid and feature-rich—it's essentially the **real-time bridge between your cars, telemetry, alerts, and frontend via WebSockets**. Here's a detailed review with some observations and suggestions:

---

1. Responsibilities

- Subscribes to car telemetry and status topics.
- Processes incoming telemetry messages and creates corresponding `Telemetry` entities.
- Checks thresholds (fuel, speed, temperature) and creates alerts if necessary.

- Broadcasts telemetry, car location, alerts, and system status via `WebSocketService`.
- Supports publishing telemetry/status messages to MQTT.
- Handles MQTT connection lifecycle (`@PostConstruct` and `@PreDestroy`).

---

2. Strengths

1. **Robust Message Handling**
    - Decodes MQTT payloads into `TelemetryDto`.
    - Properly logs errors for debugging.
    - Switch-case on message type keeps the code readable.
2. **Alert Handling**
    - Threshold checks for speed, fuel, and temperature.
    - Creates `Alert` entities and broadcasts updates.
    - Correctly handles critical alerts for admin notifications.
3. **WebSocket Integration**
    - Sends telemetry updates and car locations to clients in real-time.
    - Updates frontend when MQTT connection is lost/reconnected.
4. **Thread-Safe Subscription**
    - `AtomicBoolean subscribed` ensures you don't subscribe multiple times accidentally.
5. **Asynchronous Publishing**
    - Uses `CompletableFuture` for non-blocking MQTT publishing.
6. **Connection Management**
    - Graceful disconnect in `@PreDestroy`.
    - Handles connection loss with callback.

---

3. Observations / Suggestions

1. **Multiple Alerts Issue**
    - In `checkAndCreateAlerts()`, `createdAlert` is overwritten for each condition.
    - Right now, only the **last alert** is returned.
    - Suggestion: return a **list of alerts** so all triggered alerts are visible and broadcasted.

```java
List<Alert> createdAlerts = new ArrayList<>();
// add each alert to the list
return createdAlerts;
```

2. **Topic Parsing**
    - Currently splits topic by `/` and assumes `topicParts[1]` is always `carId`.

  - Consider **validating `carId` parsing** with `try-catch` to prevent `NumberFormatException`.
3. **Threshold Config**
   - Thresholds for fuel, speed, temperature are hardcoded.
   - Consider making them configurable via `application.properties`:

```properties
car.fuel.low=20
car.fuel.critical=10
car.speed.high=120
car.speed.critical=150
car.temp.high=50
car.temp.critical=60
```

4. **Duplicate Code**
   - Alert creation and broadcast is repeated in `checkAndCreateAlerts()` for each type.
   - Can refactor into a **helper method** like:

```java
private Alert createAndBroadcastAlert(Car car, String type, Alert.AlertSeverity severity, String message) {
    Alert alert = alertService.createAlert(car, type, severity.toString(), message);
    webSocketService.broadcastAlertUpdate(alert);
    return alert;
}
```

5. **Telemetry Timestamp**
   - Currently sets `telemetryDto.setTimestamp(LocalDateTime.now())`.
   - If the car sends its own timestamp, consider using it to avoid discrepancies.
6. **Resubscription on Reconnect**
   - If MQTT reconnects, ensure `subscribeToTopicsOnce()` is called again; currently it's called only in `@PostConstruct`.
   - Could add `connectionLost()` callback logic to resubscribe.

✅ Summary

- Overall, this is a **well-designed real-time service** integrating MQTT, database, and WebSockets.
- Minor improvements can make it:
  - Handle **multiple simultaneous alerts**.
  - Be **configurable and resilient** to connection issues.
  - Reduce **repetitive code** in alert handling.

If you want, I can **refactor `checkAndCreateAlerts()` and the MQTT subscription handling** so it can handle **multiple alerts, use configs, and auto-resubscribe** safely—making it much cleaner and production-ready.

Do you want me to do that?

```
// TelemetryService.java
package com.smartcar.monitoring.service;

import com.smartcar.monitoring.model.Telemetry;
import com.smartcar.monitoring.model.Car;
import com.smartcar.monitoring.repository.TelemetryRepository;
import com.smartcar.monitoring.repository.CarRepository;
import com.smartcar.monitoring.exception.TelemetryNotFoundException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.time.LocalDateTime;
import java.util.List;

@Service
@Transactional
public class TelemetryService {

    @Autowired
    private TelemetryRepository telemetryRepository;

    @Autowired
    private CarRepository carRepository;

    // Create new telemetry record
    public Telemetry createTelemetry(Telemetry telemetry) {
        telemetry.setCreationDate(LocalDateTime.now());
        telemetry.setLastUpdateOn(LocalDateTime.now());
        telemetry.setIsActive(true);

        if (telemetry.getTimestamp() == null) {
            telemetry.setTimestamp(LocalDateTime.now());
        }

        return telemetryRepository.save(telemetry);
    }

    // Get telemetry by ID
    public Telemetry getTelemetryById(Long id) {
        return telemetryRepository.findById(id)
                .orElseThrow(() -> new TelemetryNotFoundException("Telemetry not found with ID: " + id));
    }

    // Get all active telemetry records
    public List<Telemetry> getAllActiveTelemetry() {
        return telemetryRepository.findByIsActiveTrue();
    }

    // Get telemetry by car
    public List<Telemetry> getTelemetryByCar(Long carId) {
        return telemetryRepository.findByCarIdAndIsActiveTrue(carId);
    }

    // Get telemetry by timestamp range
    public List<Telemetry> getTelemetryByTimestampRange(LocalDateTime startTime,
LocalDateTime endTime) {
        return telemetryRepository.findByTimestampBetweenAndIsActiveTrue(startTime, endTime);
    }

    // Get telemetry by car and timestamp range
    public List<Telemetry> getTelemetryByCarAndTimestampRange(Long carId, LocalDateTime
startTime,
            LocalDateTime endTime) {
        return telemetryRepository.findByCarIdAndTimestampBetweenAndIsActiveTrue(carId,
startTime, endTime);
    }

    // Get telemetry by fuel level threshold
    public List<Telemetry> getTelemetryByFuelThreshold(Integer fuelThreshold) {
        return telemetryRepository.findByFuelLessThanAndIsActiveTrue(fuelThreshold);
    }

    // Get telemetry by temperature range
    public List<Telemetry> getTelemetryByTemperatureRange(Integer minTemp, Integer
maxTemp) {
        return telemetryRepository.findByTemperatureBetweenAndIsActiveTrue(minTemp,
maxTemp);
    }

    // Get telemetry by speed range
    public List<Telemetry> getTelemetryBySpeedRange(Integer minSpeed, Integer maxSpeed) {
        return telemetryRepository.findBySpeedBetweenAndIsActiveTrue(minSpeed, maxSpeed);
    }

    // Get telemetry by location
    public List<Telemetry> getTelemetryByLocation(String location) {
        return telemetryRepository.findByLocationContainingAndIsActiveTrue(location);
    }

    // Get latest telemetry for all cars
    public List<Telemetry> getLatestTelemetryForAllCars() {
        return telemetryRepository.findLatestTelemetryForAllCars();
    }

    // Get latest telemetry for specific car
    public List<Telemetry> getLatestTelemetryByCar(Long carId) {
        return telemetryRepository.findLatestTelemetryByCarId(carId);
    }

    // Get telemetry records created in date range
    public List<Telemetry> getTelemetryCreatedBetween(LocalDateTime startDate,
LocalDateTime endDate) {
        return telemetryRepository.findTelemetryCreatedBetween(startDate, endDate);
    }

    // Update telemetry
    public Telemetry updateTelemetry(Long id, Telemetry telemetryDetails) {
        Telemetry telemetry = getTelemetryById(id);

        telemetry.setSpeed(telemetryDetails.getSpeed());
        telemetry.setFuel(telemetryDetails.getFuel());
        telemetry.setTemperature(telemetryDetails.getTemperature());
        telemetry.setLocation(telemetryDetails.getLocation());
        telemetry.setLastUpdateOn(LocalDateTime.now());

        return telemetryRepository.save(telemetry);
    }

    // Soft delete telemetry
    public void deactivateTelemetry(Long id) {
        Telemetry telemetry = getTelemetryById(id);
        telemetry.setIsActive(false);
        telemetry.setLastUpdateOn(LocalDateTime.now());
        telemetryRepository.save(telemetry);
    }

    // Reactivate telemetry
    public void reactivateTelemetry(Long id) {
        Telemetry telemetry = getTelemetryById(id);
        telemetry.setIsActive(true);
        telemetry.setLastUpdateOn(LocalDateTime.now());
        telemetryRepository.save(telemetry);
    }

    // Count telemetry records by car
    public long countTelemetryByCar(Long carId) {
        return telemetryRepository.countByCarIdAndIsActiveTrue(carId);
    }

    // Count total active telemetry records
    public long countTotalActiveTelemetry() {
        return telemetryRepository.countByIsActiveTrue();
    }

    // Count telemetry records in time range
    public long countTelemetryInTimeRange(LocalDateTime startTime, LocalDateTime endTime) {
        return telemetryRepository.countByTimestampBetweenAndIsActiveTrue(startTime,
endTime);
    }

    // Count telemetry records with low fuel
    public long countTelemetryWithLowFuel(Integer fuelThreshold) {
        return telemetryRepository.countByFuelLessThanAndIsActiveTrue(fuelThreshold);
    }

    // Count telemetry records with high temperature
    public long countTelemetryWithHighTemperature(Integer tempThreshold) {
        return
telemetryRepository.countByTemperatureGreaterThanAndIsActiveTrue(tempThreshold);
    }

    // Count telemetry records with high speed
    public long countTelemetryWithHighSpeed(Integer speedThreshold) {
        return telemetryRepository.countBySpeedGreaterThanAndIsActiveTrue(speedThreshold);
    }

    // Get telemetry statistics for dashboard
    public TelemetryStatistics getTelemetryStatistics(Long carId, LocalDateTime startTime,
LocalDateTime endTime) {
        List<Telemetry> telemetryList = getTelemetryByCarAndTimestampRange(carId, startTime,
endTime);

        if (telemetryList.isEmpty()) {
            return new TelemetryStatistics();
        }

        TelemetryStatistics stats = new TelemetryStatistics();
        stats.setTotalRecords(telemetryList.size());

        // Calculate averages
        double avgSpeed =
telemetryList.stream().mapToInt(Telemetry::getSpeed).average().orElse(0.0);
        double avgFuel = telemetryList.stream().mapToInt(Telemetry::getFuel).average().orElse(0.0);
        double avgTemperature =
telemetryList.stream().mapToInt(Telemetry::getTemperature).average().orElse(0.0);

        stats.setAverageSpeed(Math.round(avgSpeed * 100.0) / 100.0);
        stats.setAverageFuel(Math.round(avgFuel * 100.0) / 100.0);
        stats.setAverageTemperature(Math.round(avgTemperature * 100.0) / 100.0);

        // Find min/max values
        stats.setMinSpeed(telemetryList.stream().mapToInt(Telemetry::getSpeed).min().orElse(0));
        stats.setMaxSpeed(telemetryList.stream().mapToInt(Telemetry::getSpeed).max().orElse(0));
        stats.setMinFuel(telemetryList.stream().mapToInt(Telemetry::getFuel).min().orElse(0));
        stats.setMaxFuel(telemetryList.stream().mapToInt(Telemetry::getFuel).max().orElse(0));

stats.setMinTemperature(telemetryList.stream().mapToInt(Telemetry::getTemperature).min().orE
lse(0));

stats.setMaxTemperature(telemetryList.stream().mapToInt(Telemetry::getTemperature).max().or
Else(0));

        return stats;
    }

    // Inner class for telemetry statistics
    public static class TelemetryStatistics {
        private int totalRecords;
        private double averageSpeed;
        private double averageFuel;
        private double averageTemperature;
        private int minSpeed;
        private int maxSpeed;
        private int minFuel;
        private int maxFuel;
        private int minTemperature;
        private int maxTemperature;

        // Getters and setters
        public int getTotalRecords() {
            return totalRecords;
        }

        public void setTotalRecords(int totalRecords) {
            this.totalRecords = totalRecords;
        }

        public double getAverageSpeed() {
            return averageSpeed;
        }

        public void setAverageSpeed(double averageSpeed) {
            this.averageSpeed = averageSpeed;
        }

        public double getAverageFuel() {
            return averageFuel;
        }

        public void setAverageFuel(double averageFuel) {
            this.averageFuel = averageFuel;
        }

        public double getAverageTemperature() {
            return averageTemperature;
        }

        public void setAverageTemperature(double averageTemperature) {
```

```
        this.averageTemperature = averageTemperature;
    }

    public int getMinSpeed() {
        return minSpeed;
    }

    public void setMinSpeed(int minSpeed) {
        this.minSpeed = minSpeed;
    }

    public int getMaxSpeed() {
        return maxSpeed;
    }

    public void setMaxSpeed(int maxSpeed) {
        this.maxSpeed = maxSpeed;
    }

    public int getMinFuel() {
        return minFuel;
    }

    public void setMinFuel(int minFuel) {
        this.minFuel = minFuel;
    }

    public int getMaxFuel() {
        return maxFuel;
    }

    public void setMaxFuel(int maxFuel) {
        this.maxFuel = maxFuel;
    }

    public int getMinTemperature() {
        return minTemperature;
    }

    public void setMinTemperature(int minTemperature) {
        this.minTemperature = minTemperature;
    }

    public int getMaxTemperature() {
        return maxTemperature;
    }

    public void setMaxTemperature(int maxTemperature) {
        this.maxTemperature = maxTemperature;
    }
    }
}
```

Looking at your `TelemetryService.java`, this is a **comprehensive service for handling telemetry data** with proper CRUD operations, queries, statistics, and soft deletion. Here's a detailed review and some suggestions:

---

1. Responsibilities

- CRUD operations for `Telemetry` records.
- Query telemetry by car, timestamp, speed, fuel, temperature, and location.
- Compute telemetry statistics (min, max, average) for dashboards.
- Soft delete/reactivate telemetry records.
- Count telemetry records based on various filters.

---

2. Strengths

1. **Well-structured**
    - Clear separation of CRUD operations, queries, and statistics.
    - Follows Spring conventions (`@Service`, `@Transactional`).
2. **Flexible Querying**
    - Can query by car, time range, fuel, temperature, speed, and location.
    - Provides both list and count-based queries.
3. **Statistics Computation**
    - Calculates min, max, and averages.
    - Rounds averages to two decimal places for dashboard display.
4. **Soft Deletion**
    - Uses `isActive` flag, consistent with your `CarService` and `DriverService`.
5. **Consistency**
    - Auto sets timestamps (`creationDate`, `lastUpdateOn`) if not provided.
    - Ensures `Telemetry` is always associated with a `Car`.

---

3. Observations / Suggestions

1. **Redundant Repository Calls**
    - Many queries are repeated like `findByCarIdAndIsActiveTrue` or `countByCarIdAndIsActiveTrue`.
    - Could potentially combine queries in repository with optional filters to reduce repetitive code.
2. **Handling Empty Lists**

  - Statistics computation currently returns zeroes if no telemetry exists, which is fine.
  - Just make sure dashboard handles this gracefully.

3. **Timestamp Consistency**
   - When creating telemetry in `createTelemetry()`, you set `LocalDateTime.now()` if `timestamp` is null.
   - In real-time telemetry from MQTT, you may want to **respect the car's reported timestamp** if available.
4. **Potential Performance Issue**
   - For `getTelemetryStatistics()`, fetching **all telemetry records** for a car in a time range may be heavy if the dataset is large.
   - Consider using **repository queries with aggregation (avg, min, max)** to push computation to the database.

   Example:

```java
@Query("SELECT new com.smartcar.monitoring.dto.TelemetryStatsDTO(
           COUNT(t), AVG(t.speed), AVG(t.fuel), AVG(t.temperature),
           MIN(t.speed), MAX(t.speed),
           MIN(t.fuel), MAX(t.fuel),
           MIN(t.temperature), MAX(t.temperature))
       FROM Telemetry t
       WHERE t.car.id = :carId AND t.timestamp BETWEEN :start AND :end AND t.isActive =
true")
TelemetryStatsDTO getTelemetryStats(@Param("carId") Long carId, @Param("start")
LocalDateTime start, @Param("end") LocalDateTime end);
```

   This avoids fetching huge lists into memory.

5. **DTOs for Statistics**
   - Currently, you return an inner class `TelemetryStatistics`.
   - Consider creating a separate DTO to decouple persistence layer from API layer.
6. **Validation**
   - For setters like `setSpeed`, `setFuel`, `setTemperature`, you could add **range validation** to ensure telemetry is sane before saving.

✅ Summary

- `TelemetryService` is **robust, clear, and consistent** with your other services.
- Minor improvements:
  - Optimize **statistics queries** for performance.
  - Use **DTOs** for cleaner API responses.
  - Respect incoming telemetry timestamps from cars.
  - Optional: add **range validation** for telemetry fields.

If you want, I can **refactor `getTelemetryStatistics()` and related queries** to be **DB-optimized, handle large datasets efficiently, and return a ready-to-use DTO** for the dashboard.

Do you want me to do that?

```
// UserService.java
package com.smartcar.monitoring.service;

import com.smartcar.monitoring.model.User;
import com.smartcar.monitoring.model.User.UserRole;
import com.smartcar.monitoring.repository.UserRepository;
import com.smartcar.monitoring.exception.UserNotFoundException;
import com.smartcar.monitoring.exception.UserAlreadyExistsException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.time.LocalDateTime;
import java.util.List;
import java.util.Optional;

@Service
@Transactional
public class UserService {

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private PasswordEncoder passwordEncoder;

    // Create new user
    public User createUser(User user) {
        // Check if username already exists
        if (userRepository.existsByUsername(user.getUsername())) {
            throw new UserAlreadyExistsException("Username already exists: " +
user.getUsername());
        }

        // Check if email already exists
        if (userRepository.existsByEmail(user.getEmail())) {
            throw new UserAlreadyExistsException("Email already exists: " + user.getEmail());
        }

        // Check if contact number already exists
        if (userRepository.existsByContactNumber(user.getContactNumber())) {
            throw new UserAlreadyExistsException("Contact number already exists: " +
user.getContactNumber());
```

```
        }

        // Encode password
        user.setPassword(passwordEncoder.encode(user.getPassword()));

        // Set creation date and active status
        user.setCreationDate(LocalDateTime.now());
        user.setLastUpdateOn(LocalDateTime.now());
        user.setIsActive(true);

        return userRepository.save(user);
    }

    // Get user by ID
    public User getUserById(Long id) {
        return userRepository.findById(id)
                .orElseThrow(() -> new UserNotFoundException("User not found with ID: " + id));
    }

    // Get user by username (for authentication)
    public Optional<User> getUserByUsername(String username) {
        return userRepository.findByUsername(username);
    }

    // Get user by email
    public Optional<User> getUserByEmail(String email) {
        return userRepository.findByEmail(email);
    }

    // Get all active users
    public List<User> getAllActiveUsers() {
        return userRepository.findByIsActiveTrue();
    }

    // Get users by role
    public List<User> getUsersByRole(UserRole role) {
        return userRepository.findByRoleAndIsActiveTrue(role);
    }

    // Update user
    public User updateUser(Long id, User userDetails) {
        User user = getUserById(id);

        // Check if new username conflicts (if changed)
        if (!user.getUsername().equals(userDetails.getUsername()) &&
                userRepository.existsByUsername(userDetails.getUsername())) {
            throw new UserAlreadyExistsException("Username already exists: " +
userDetails.getUsername());
        }

        // Check if new email conflicts (if changed)
        if (!user.getEmail().equals(userDetails.getEmail()) &&
                userRepository.existsByEmail(userDetails.getEmail())) {
            throw new UserAlreadyExistsException("Email already exists: " + userDetails.getEmail());
        }

        // Check if new contact number conflicts (if changed)
        if (!user.getContactNumber().equals(userDetails.getContactNumber()) &&
                userRepository.existsByContactNumber(userDetails.getContactNumber())) {
            throw new UserAlreadyExistsException("Contact number already exists: " +
userDetails.getContactNumber());
        }

        // Update fields
        user.setName(userDetails.getName());
        user.setAge(userDetails.getAge());
        user.setGender(userDetails.getGender());
        user.setContactNumber(userDetails.getContactNumber());
        user.setEmail(userDetails.getEmail());
        user.setRole(userDetails.getRole());
        user.setLastUpdateOn(LocalDateTime.now());

        // Update password only if provided
        if (userDetails.getPassword() != null && !userDetails.getPassword().isEmpty()) {
            user.setPassword(passwordEncoder.encode(userDetails.getPassword()));
        }

        return userRepository.save(user);
    }

    // Soft delete user (mark as inactive)
    public void deactivateUser(Long id) {
        User user = getUserById(id);
        user.setIsActive(false);
        user.setLastUpdateOn(LocalDateTime.now());
        userRepository.save(user);
    }

    // Reactivate user
    public void reactivateUser(Long id) {
        User user = getUserById(id);
        user.setIsActive(true);
        user.setLastUpdateOn(LocalDateTime.now());
        userRepository.save(user);
    }

    // Get users by age range
    public List<User> getUsersByAgeRange(Integer minAge, Integer maxAge) {
        return userRepository.findByAgeBetweenAndIsActiveTrue(minAge, maxAge);
    }

    // Get users by gender
    public List<User> getUsersByGender(User.Gender gender) {
        return userRepository.findByGenderAndIsActiveTrue(gender);
    }

    // Get users created in date range
    public List<User> getUsersCreatedBetween(LocalDateTime startDate, LocalDateTime endDate) {
        return userRepository.findUsersCreatedBetween(startDate, endDate);
    }

    // Get users by name pattern
    public List<User> getUsersByNamePattern(String namePattern) {
        return userRepository.findUsersByNamePattern(namePattern);
    }

    // Count active users by role
    public long countActiveUsersByRole(UserRole role) {
        return userRepository.countByRoleAndIsActiveTrue(role);
    }

    // Count total active users
    public long countTotalActiveUsers() {
        return userRepository.countByIsActiveTrue();
    }

    // Validate user credentials
    public boolean validateCredentials(String username, String rawPassword) {
        Optional<User> userOpt = getUserByUsername(username);
        if (userOpt.isPresent()) {
            User user = userOpt.get();
            return user.getIsActive() && passwordEncoder.matches(rawPassword, user.getPassword());
        }
        return false;
    }
}
```

Looking at your `UserService.java`, this is a **well-structured Spring service for user management**. Here's a detailed review with insights and potential improvements:

---

1. Responsibilities

- CRUD operations for `User`.
- Soft deletion/reactivation of users.
- Query users by role, age, gender, name pattern, creation date.
- Count users by role or total active users.
- Validate credentials for authentication.

- Prevent duplication of username, email, and contact number.

---

2. Strengths

1. **Validation & Uniqueness**
    - Checks for existing username, email, and contact number both during creation and update.
    - Ensures data integrity and avoids duplicate users.
2. **Password Handling**
    - Properly encodes password using `PasswordEncoder`.
    - Updates password only if a new one is provided during updates.
3. **Soft Deletion**
    - Uses `isActive` flag to deactivate/reactivate users.
    - Consistent with `DriverService` and `TelemetryService`.
4. **Flexible Queries**
    - Supports querying by age, gender, role, name pattern, and creation date.
5. **Credential Validation**
    - Checks both password and `isActive` flag for authentication.

---

3. Observations / Suggestions

1. **Update Checks Could Be Optimized**
    - Currently, for every field (`username`, `email`, `contactNumber`), you check for existence before updating.
    - For large datasets, consider fetching any conflicting user in **one query** instead of three separate existence checks.
2. **Password Handling**
    - Currently, empty password strings are ignored. Ensure the front-end never sends empty strings unintentionally.
    - Optional: Consider a `changePassword` method separately for clarity.
3. **DTO Usage**
    - Service directly takes and returns `User` entities.
    - For APIs, it might be better to use **DTOs** to avoid exposing sensitive info (like password) in responses.
4. **Performance**
    - Queries like `findUsersByNamePattern` and `findUsersCreatedBetween` could be heavy on large datasets. Ensure proper indexing on `name` and `creationDate`.
5. **Additional Security**
    - Could enforce stronger password policies (length, complexity) during creation/update.

6. **Transaction Handling**
    - Good use of `@Transactional` for consistency.

✅ Summary

`UserService` is **robust and comprehensive**, aligned with your other services (`DriverService`, `TelemetryService`).

- Strengths: data integrity, soft deletion, flexible queries, password encoding.
- Improvements: optimize existence checks during updates, use DTOs for API responses, consider indexing and password policy.

If you want, I can **suggest a refactored version of `UserService`** that:

- Uses **DTOs** for input/output.
- Consolidates uniqueness checks.
- Improves password handling and security.

Do you want me to do that?

```
// WebSocketService.java
package com.smartcar.monitoring.service;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.smartcar.monitoring.dto.TelemetryDto;
import com.smartcar.monitoring.model.Alert;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.messaging.simp.SimpMessagingTemplate;
import org.springframework.stereotype.Service;

import java.time.LocalDateTime;
import java.util.HashMap;
import java.util.Map;

@Service
public class WebSocketService {

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

    @Autowired
    private SimpMessagingTemplate messagingTemplate;

    @Autowired
    private ObjectMapper objectMapper;
```

```
// Send telemetry updates to all connected clients
public void broadcastTelemetryUpdate(TelemetryDto telemetryDto) {
    try {
        Map<String, Object> message = new HashMap<>();
        message.put("type", "TELEMETRY_UPDATE");
        message.put("data", telemetryDto);
        message.put("timestamp", LocalDateTime.now());

        messagingTemplate.convertAndSend("/topic/telemetry", message);
        logger.debug("Broadcasted telemetry update for car: {}", telemetryDto.getCarId());
    } catch (Exception e) {
        logger.error("Error broadcasting telemetry update", e);
    }
}

// Send telemetry updates to specific car subscribers
public void sendTelemetryToCar(Long carId, TelemetryDto telemetryDto) {
    try {
        Map<String, Object> message = new HashMap<>();
        message.put("type", "CAR_TELEMETRY");
        message.put("carId", carId);
        message.put("data", telemetryDto);
        message.put("timestamp", LocalDateTime.now());

        messagingTemplate.convertAndSend("/topic/car/" + carId + "/telemetry", message);
        logger.debug("Sent telemetry update to car {} subscribers", carId);
    } catch (Exception e) {
        logger.error("Error sending telemetry to car {}", carId, e);
    }
}

// Send alert updates to all connected clients
public void broadcastAlertUpdate(Alert alert) {
    try {
        Map<String, Object> message = new HashMap<>();
        message.put("type", "ALERT_UPDATE");
        message.put("data", alert);
        message.put("timestamp", LocalDateTime.now());

        messagingTemplate.convertAndSend("/topic/alerts", message);
        logger.debug("Broadcasted alert update: {}", alert.getType());
    } catch (Exception e) {
        logger.error("Error broadcasting alert update", e);
    }
}

// Send critical alerts to admin dashboard
public void sendCriticalAlertToAdmins(Alert alert) {
    try {
        Map<String, Object> message = new HashMap<>();
```

```
            message.put("type", "CRITICAL_ALERT");
            message.put("data", alert);
            message.put("timestamp", LocalDateTime.now());

            messagingTemplate.convertAndSend("/topic/admin/critical-alerts", message);
            logger.debug("Sent critical alert to admin dashboard: {}", alert.getType());
        } catch (Exception e) {
            logger.error("Error sending critical alert to admins", e);
        }
    }

    // Send system status updates
    public void broadcastSystemStatus(Map<String, Object> status) {
        try {
            Map<String, Object> message = new HashMap<>();
            message.put("type", "SYSTEM_STATUS");
            message.put("data", status);
            message.put("timestamp", LocalDateTime.now());

            messagingTemplate.convertAndSend("/topic/system/status", message);
            logger.debug("Broadcasted system status update");
        } catch (Exception e) {
            logger.error("Error broadcasting system status", e);
        }
    }

    // Send MQTT connection status
    public void broadcastMqttStatus(boolean connected) {
        try {
            Map<String, Object> message = new HashMap<>();
            message.put("type", "MQTT_STATUS");
            message.put("connected", connected);
            message.put("timestamp", LocalDateTime.now());

            messagingTemplate.convertAndSend("/topic/system/mqtt-status", message);
            logger.debug("Broadcasted MQTT status: {}", connected);
        } catch (Exception e) {
            logger.error("Error broadcasting MQTT status", e);
        }
    }

    // Send simulator status updates
    public void broadcastSimulatorStatus(boolean running) {
        try {
            Map<String, Object> message = new HashMap<>();
            message.put("type", "SIMULATOR_STATUS");
            message.put("running", running);
            message.put("timestamp", LocalDateTime.now());

            messagingTemplate.convertAndSend("/topic/system/simulator-status", message);
            logger.debug("Broadcasted simulator status: {}", running);
        } catch (Exception e) {
            logger.error("Error broadcasting simulator status", e);
        }
    }

    // Send dashboard statistics updates
    public void broadcastDashboardStats(Map<String, Object> stats) {
        try {
            Map<String, Object> message = new HashMap<>();
            message.put("type", "DASHBOARD_STATS");
            message.put("data", stats);
            message.put("timestamp", LocalDateTime.now());

            messagingTemplate.convertAndSend("/topic/dashboard/stats", message);
            logger.debug("Broadcasted dashboard statistics");
        } catch (Exception e) {
            logger.error("Error broadcasting dashboard stats", e);
        }
    }

    // Send user-specific notifications
    public void sendNotificationToUser(String username, String message) {
        try {
            Map<String, Object> notification = new HashMap<>();
            notification.put("type", "NOTIFICATION");
            notification.put("message", message);
            notification.put("timestamp", LocalDateTime.now());

            messagingTemplate.convertAndSendToUser(username, "/queue/notifications",
notification);
            logger.debug("Sent notification to user: {}", username);
        } catch (Exception e) {
            logger.error("Error sending notification to user {}", username, e);
        }
    }

    // Send real-time car location updates
    public void broadcastCarLocation(Long carId, String location) {
        try {
            Map<String, Object> message = new HashMap<>();
            message.put("type", "CAR_LOCATION");
            message.put("carId", carId);
            message.put("location", location);
            message.put("timestamp", LocalDateTime.now());

            messagingTemplate.convertAndSend("/topic/car/" + carId + "/location", message);
            messagingTemplate.convertAndSend("/topic/map/locations", message);
            logger.debug("Broadcasted car location update for car: {}", carId);
        } catch (Exception e) {
            logger.error("Error broadcasting car location for car {}", carId, e);
        }
    }

    // Send heartbeat/ping to keep connections alive
    public void sendHeartbeat() {
        try {
            Map<String, Object> heartbeat = new HashMap<>();
            heartbeat.put("type", "HEARTBEAT");
            heartbeat.put("timestamp", LocalDateTime.now());

            messagingTemplate.convertAndSend("/topic/heartbeat", heartbeat);
        } catch (Exception e) {
            logger.error("Error sending heartbeat", e);
        }
    }
}
```

Looking at your `WebSocketService.java`, this class is **well-structured and comprehensive** for real-time updates in your smart car monitoring system. Here's a detailed review and observations:

---

1. Responsibilities

- Broadcasting real-time telemetry, alerts, system status, and dashboard statistics.
- Sending notifications to specific users.
- Sending MQTT and simulator status updates.
- Sending car location updates to both car-specific and map topics.
- Heartbeat mechanism to keep connections alive.

---

2. Strengths

1. **Consistency**
    - All messages follow a consistent structure: type, data, timestamp.
    - Makes it easy for front-end clients to parse.
2. **Coverage**
    - Supports all major real-time events: telemetry, alerts, system status, simulator status, dashboard stats, user notifications, car location, heartbeat.
3. **Error Handling**
    - Every method catches exceptions and logs errors, preventing crashes on WebSocket failures.
4. **Reusability**
    - Can be injected and used across services like `MqttService`, `TelemetryService`, `AlertService`.
5. **Granularity**
    - Supports both global broadcasts and user-specific messages.
    - Sends car-specific telemetry and location updates to specific subscribers.

3. Observations / Suggestions

1. **Message Structure**
    - Currently, messages are built manually in every method. Consider creating a **helper method** to build the standard message structure to reduce code duplication:

```java
private Map<String, Object> buildMessage(String type, Object data) {
    Map<String, Object> message = new HashMap<>();
    message.put("type", type);
    message.put("data", data);
    message.put("timestamp", LocalDateTime.now());
    return message;
}
```

This can simplify broadcasting calls:

```java
messagingTemplate.convertAndSend("/topic/alerts", buildMessage("ALERT_UPDATE", alert));
```

2. **Performance**
    - For high-frequency telemetry and location updates, broadcasting to multiple topics can become expensive.
    - Consider **batching updates** or using **topics with wildcard subscriptions** to reduce overhead.
3. **Heartbeat**
    - Currently, no mechanism to detect missed heartbeats or disconnect inactive clients. You could enhance this with timestamps or client-side pings.
4. **Security**
    - `convertAndSendToUser` assumes Spring Security user mapping is correctly set up.
    - Ensure proper authorization on endpoints to avoid leaking notifications.
5. **Optional Enhancement**
    - Could include **QoS levels** for critical messages (like alerts) if using WebSocket over MQTT bridges or similar.

✅ Summary

`WebSocketService` is **robust, comprehensive, and aligned with your MQTT and telemetry services**.

- Strengths: consistent messaging, full coverage of real-time events, error handling.
- Improvements: reduce code duplication with a helper for message building, optimize high-frequency broadcasts, enhance heartbeat monitoring.

If you want, I can **refactor this service** to:

- Use a **central `buildMessage` helper**.
- Optimize **car location and telemetry broadcasting** for better performance.
- Keep the code clean and DRY.

Do you want me to do that?

```
// TelemetrySimulator.java
package com.smartcar.monitoring.simulator;

import com.smartcar.monitoring.dto.TelemetryDto;
import com.smartcar.monitoring.model.Car;
import com.smartcar.monitoring.service.CarService;
import com.smartcar.monitoring.service.MqttService;
import com.smartcar.monitoring.service.WebSocketService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import jakarta.annotation.PostConstruct;
import java.time.LocalDateTime;
import java.util.List;
import java.util.Random;
import java.util.concurrent.atomic.AtomicBoolean;

@Component
public class TelemetrySimulator {

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

    @Autowired
    private MqttService mqttService;

    @Autowired
    private CarService carService;

    @Autowired
    private WebSocketService webSocketService;

    @Value("${simulator.enabled:true}")
    private boolean simulatorEnabled;

    @Value("${simulator.interval:5000}")
    private long simulatorInterval;

    @Value("${simulator.car.count:5}")
    private int carCount;

    private final Random random = new Random();
    private final AtomicBoolean isRunning = new AtomicBoolean(false);

    // Location coordinates for simulation (major cities)
    private final String[] locations = {
            "New York, NY", "Los Angeles, CA", "Chicago, IL", "Houston, TX", "Phoenix, AZ",
            "Philadelphia, PA", "San Antonio, TX", "San Diego, CA", "Dallas, TX", "San Jose, CA"
    };

    // Car statuses for simulation
    private final String[] carStatuses = { "IDLE", "MOVING", "PARKED", "MAINTENANCE" };

    @PostConstruct
    public void init() {
        if (simulatorEnabled) {
            logger.info("Telemetry Simulator initialized. Interval: {}ms, Car Count: {}",
simulatorInterval, carCount);
            startSimulation();
        } else {
            logger.info("Telemetry Simulator is disabled");
        }
    }

    @Scheduled(fixedDelayString = "${simulator.interval:5000}")
    public void simulateTelemetry() {
        if (!simulatorEnabled || !isRunning.get()) {
            return;
        }

        try {
            List<Car> activeCars = carService.getAllActiveCars();

            if (activeCars.isEmpty()) {
                logger.warn("No active cars found for simulation");
                return;
            }

            // Simulate telemetry for each active car
            for (Car car : activeCars) {
                if (car.getIsActive()) {
                    TelemetryDto telemetryDto = generateTelemetryData(car);
                    mqttService.publishTelemetry(car.getId(), telemetryDto);

                    // Update car status occasionally
                    if (random.nextInt(10) == 0) {
                        String newStatus = carStatuses[random.nextInt(carStatuses.length)];
                        mqttService.publishStatus(car.getId(), newStatus);
                    }
                }
            }

        } catch (Exception e) {
            logger.error("Error in telemetry simulation", e);
        }
    }

    private TelemetryDto generateTelemetryData(Car car) {
        TelemetryDto telemetryDto = new TelemetryDto();

        // Generate realistic speed (0-140 km/h)
        int speed = generateSpeed();
        telemetryDto.setSpeed(speed);

        // Generate fuel level (0-100%)
        int fuelLevel = generateFuelLevel();
        telemetryDto.setFuelLevel(fuelLevel);

        // Generate temperature (-10 to 70°C)
        int temperature = generateTemperature();
        telemetryDto.setTemperature(temperature);

        // Generate location
        String location = generateLocation();
        telemetryDto.setLocation(location);

        // Set timestamp
        telemetryDto.setTimestamp(LocalDateTime.now());

        return telemetryDto;
    }

    private int generateSpeed() {
        // 70% chance of normal speed (0-80 km/h), 25% chance of highway speed (80-120
        // km/h), 5% chance of high speed (120-140 km/h)
        double chance = random.nextDouble();

        if (chance < 0.70) {
            return random.nextInt(81); // 0-80 km/h
        } else if (chance < 0.95) {
            return 80 + random.nextInt(41); // 80-120 km/h
        } else {
            return 120 + random.nextInt(21); // 120-140 km/h
        }
    }

    private int generateFuelLevel() {
        // 60% chance of normal fuel (20-100%), 30% chance of low fuel (10-30%), 10%
        // chance of very low fuel (5-15%)
        double chance = random.nextDouble();
        if (chance < 0.60) {
            return 20 + random.nextInt(81); // 20-100%
        } else if (chance < 0.90) {
            return 10 + random.nextInt(21); // 10-30%
        } else {
            return 5 + random.nextInt(11); // 5-15%
        }
    }

    private int generateTemperature() {
        // 80% chance of normal temperature (10-40°C), 15% chance of extreme temperature
        // (40-70°C), 5% chance of cold (-10 to 10°C)
        double chance = random.nextDouble();

        if (chance < 0.80) {
            return 10 + random.nextInt(31); // 10-40°C
        } else if (chance < 0.95) {
            return 40 + random.nextInt(31); // 40-70°C
        } else {
            return -10 + random.nextInt(21); // -10 to 10°C
        }
    }

    private String generateLocation() {
        return locations[random.nextInt(locations.length)];
    }

    public void startSimulation() {
        if (isRunning.compareAndSet(false, true)) {
            logger.info("Telemetry simulation started");
            webSocketService.broadcastSimulatorStatus(true);
        }
    }

    public void stopSimulation() {
        if (isRunning.compareAndSet(true, false)) {
            logger.info("Telemetry simulation stopped");
            webSocketService.broadcastSimulatorStatus(false);
        }
    }

    public boolean isRunning() {
        return isRunning.get();
    }

    public void setSimulatorEnabled(boolean enabled) {
        this.simulatorEnabled = enabled;
        if (enabled) {
            startSimulation();
        } else {
            stopSimulation();
        }
        logger.info("Telemetry simulator enabled: {}", enabled);
    }

    public void setSimulatorInterval(long interval) {
        this.simulatorInterval = interval;
        logger.info("Telemetry simulator interval updated: {}ms", interval);
    }
}
```

Your `TelemetrySimulator.java` is **well-structured** and effectively simulates telemetry data for active cars. Here's a thorough review and some recommendations:

---

1. Responsibilities

- Simulates telemetry data (speed, fuel, temperature, location) for active cars.
- Publishes telemetry and occasional car status updates via `MqttService`.
- Provides start/stop control and broadcasts simulator status via `WebSocketService`.
- Configurable via Spring properties (`simulator.enabled`, `simulator.interval`, `simulator.car.count`).

---

2. Strengths

1. **Realistic Simulation**
    - Probabilistic generation of speed, fuel, and temperature makes data look realistic.
    - Different ranges for normal, low, and extreme values.
2. **Integration**
    - Works seamlessly with `MqttService` and `WebSocketService`.
    - Uses active cars from `CarService`, keeping simulation dynamic.
3. **Configurability**
    - Interval, simulator enable/disable, and car count are all configurable.
    - Can adjust behavior without code changes.
4. **Thread Safety**
    - Uses `AtomicBoolean` for `isRunning` to safely start/stop the simulation.
5. **Logging**
    - Clear logging for simulation start, stop, and errors.

---

3. Observations / Suggestions

1. **Random Generation**
    - Current method uses simple `Random`. For more realistic simulations, you could use **Gaussian distributions** for speed/fuel/temperature to mimic natural variations.

2. **Location Simulation**
    - Locations are currently hard-coded cities.
    - If needed, you could simulate **movement over time** by slightly modifying latitude/longitude instead of jumping between cities.
3. **Scheduled Simulation**
    - Uses `@Scheduled(fixedDelayString)`, which is fine.
    - Ensure that `simulateTelemetry` does not take longer than `simulatorInterval` for large car counts to avoid overlapping executions.
4. **Telemetry Timestamps**
    - Currently, `TelemetryDto.setTimestamp(LocalDateTime.now())` for each generation.
    - If simulating delayed updates or network lag, you could randomize timestamp offsets.
5. **Status Updates**
    - Currently, `1/10` chance to update status.
    - You could make this probability configurable for flexibility.
6. **Scalability**
    - For a large number of cars, consider parallelizing telemetry generation with `CompletableFuture` or thread pools to prevent blocking.
7. **DRY / Refactor**
    - The generation methods (`generateSpeed`, `generateFuelLevel`, `generateTemperature`) have similar probability patterns. Could be refactored into a **generic probabilistic range generator** to reduce repetition.

---

4. Optional Enhancements

- Add **acceleration/deceleration simulation** for more realistic speed changes over time.
- Include **battery/electric car metrics** if your system supports EVs.
- Simulate **alerts** directly in the simulator (like low fuel or high temperature) for testing the alert system end-to-end.

✅ Summary

Your simulator is **solid, realistic, and integrates well with MQTT/WebSocket services**.
Main improvements could focus on:

- DRY refactoring of random generators.
- Configurable probabilities and movement simulation.
- Performance/scalability for larger fleets.

If you want, I can **refactor** `TelemetrySimulator` to:

- Make probability-based generators generic.
- Make location and status updates more realistic.
- Optimize for large fleets with parallel updates.

Do you want me to do that?

```
# application.properties
spring.application.name=smart-car-monitoring

# Server Configuration
server.port=8080

# Database Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/smartcar
spring.datasource.username=root
spring.datasource.password=1234
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# JPA Configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

# JWT Configuration
jwt.secret=smartCarMonitoringSecretKey2025ForJWTTokenGeneration
jwt.expiration=86400000

# MQTT Configuration - HiveMQ Cloud
mqtt.broker.url=ssl://28f28b09486648cc85d19ad078bcfe44.s1.eu.hivemq.cloud:8883
mqtt.broker.websocket.url=wss://28f28b09486648cc85d19ad078bcfe44.s1.eu.hivemq.cloud:8884/mqtt
mqtt.client.id=smart-car-backend-${random.uuid}
mqtt.username=SmartCar
mqtt.password=SmartCar123
mqtt.topic.prefix=car
mqtt.connection.timeout=30
mqtt.keep.alive.interval=60
mqtt.clean.session=true
mqtt.auto.reconnect=true

# WebSocket Configuration
websocket.endpoint=/ws
websocket.allowed-origins=*

# Simulator Configuration
simulator.enabled=true
simulator.interval=5000
simulator.car.count=5

# Logging Configuration
logging.level.com.smartcar.monitoring=DEBUG
logging.level.org.springframework.security=DEBUG
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
logging.level.org.eclipse.paho.client.mqttv3=DEBUG
```

Your `application.properties` looks **well-structured** and covers the full configuration for your Smart Car Monitoring project. Here's a detailed review and some recommendations:

---

1. Server & Application

```properties
spring.application.name=smart-car-monitoring
server.port=8080
```

✅ Clear naming and server port configuration.
💡 Optional: You could externalize the port via environment variables for flexibility in different environments.

---

2. Database Configuration

```properties
spring.datasource.url=jdbc:mysql://localhost:3306/smartcar
spring.datasource.username=root
spring.datasource.password=1234
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

✅ Proper JDBC + JPA configuration.
💡 **Security note:** Avoid storing DB credentials in plain text in production. Use environment variables or Spring Cloud Config.

---

3. JWT Configuration

```properties
jwt.secret=smartCarMonitoringSecretKey2025ForJWTTokenGeneration
jwt.expiration=86400000
```

✅ Secret key and expiration defined.
💡 Optional: Use a stronger key from environment variables and rotate periodically.

---

4. MQTT Configuration

```properties
mqtt.broker.url=ssl://28f28b09486648cc85d19ad078bcfe44.s1.eu.hivemq.cloud:8883
mqtt.broker.websocket.url=wss://28f28b09486648cc85d19ad078bcfe44.s1.eu.hivemq.cloud:8884/mqtt
mqtt.client.id=smart-car-backend-${random.uuid}
mqtt.username=SmartCar
mqtt.password=SmartCar123
mqtt.topic.prefix=car
mqtt.connection.timeout=30
mqtt.keep.alive.interval=60
mqtt.clean.session=true
mqtt.auto.reconnect=true
```

✅ Complete HiveMQ Cloud configuration.
💡 **Security note:** Password in plain text is fine for local dev, but should be stored securely in production (env vars, vault).

💡 `mqtt.client.id` uses `${random.uuid}` which is good to avoid client ID conflicts.

---

5. WebSocket Configuration

```properties
websocket.endpoint=/ws
websocket.allowed-origins=*
```

✅ Correct basic WebSocket setup.
💡 Optional: In production, restrict `allowed-origins` instead of `*` to prevent cross-origin issues.

---

6. Simulator Configuration

```properties
simulator.enabled=true
simulator.interval=5000
simulator.car.count=5
```

✅ Matches your `TelemetrySimulator` defaults.

💡 Optional: Could externalize min/max values for speed, fuel, temperature for more flexible simulation.

---

7. Logging Configuration

```properties
logging.level.com.smartcar.monitoring=DEBUG
logging.level.org.springframework.security=DEBUG
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
logging.level.org.eclipse.paho.client.mqttv3=DEBUG
```

✅ Very detailed logging for development/debugging.

💡 Optional: For production, set `INFO` or `WARN` to reduce log volume.

---

8. Overall Observations

- Properties are **well-organized by function**: server, DB, JWT, MQTT, WebSocket, simulator, logging.
- Ready for local development, HiveMQ, MySQL, and simulator integration.
- Minor improvements: security for secrets, production-friendly logging, and restricting CORS origins.

If you want, I can **suggest an improved `application.properties` setup for production** with **secure secrets, env variable placeholders, and flexible simulator configs**.

Do you want me to do that?