Service Booking & Workshop Management System (S.B.W.M.S.)

1) Story

A fast-growing automobile service chain, **AutoCare Pro**, operates 5 service centers across the city. They currently manage:

- Service bookings
- Customer vehicle records
- Mechanic assignments
- Spare part inventory
- Billing and payments

using Excel sheets and manual registers.

This causes issues:

- Double booking of mechanics
- Missing service history
- Incorrect spare part usage tracking
- No daily revenue visibility
- No performance tracking of mechanics

AutoCare Pro wants a Java-based Workshop Management System that uses a relational DB, supports role-based actions (Admin vs Service Advisor), logs all operations, and produces basic business reports.

You are hired as the backend team to build the first working version.

---

2) What you will build

A console-based Java application (no UI frameworks required) that:

- Manages Customers, Vehicles, Service Bookings, Mechanics, Spare Parts, Invoices, and Payments
- Persists data in MySQL / PostgreSQL using SQL + JDBC
- Uses layered architecture
- Has JUnit tests for core services/validations
- Uses Log4j logging for audit/debug

- Uses Git with proper branching + commit discipline

---

3) Mandatory Tech/Concept Coverage

Core Java (must use)

- OOP (encapsulation, abstraction, interfaces)
- Collections (List/Map usage in service layer + validations)
- Enums (roles, booking status, payment mode/status)
- Java Time API (LocalDateTime for timestamps)
- Custom exceptions + proper handling
- Input validation and defensive programming

SQL + JDBC (must use)

- Normalized schema (min 3NF where reasonable)
- CRUD using PreparedStatement
- Transactions (commit/rollback) in invoice generation
- Joins & aggregations for reporting queries
- Constraints: PK, FK, UNIQUE, NOT NULL, CHECK (if supported)

Git (must use)

- Feature branches
- Meaningful commits
- PR-style workflow even if solo (merge feature branch to main)

JUnit (must use)

- Unit tests for service-layer logic
- DAO tests optional (bonus)
- Assertions verifying behavior + edge cases

Log4j (must use)

- Separate loggers for app flow + DB errors (or separate categories)
- Levels: INFO/WARN/ERROR/DEBUG
- Log file output + console output
- No System.out.println() for application flow (only menu prompts / final display outputs)

Architecture + Quality (must use)

- Layered design + clean separation
- Centralized exception strategy
- No SQL inside service classes (DAO only)
- Config via properties file
- Code quality: naming, SRP, no massive classes

---

4) Functional Requirements

A) User Roles

ADMIN

- Add/update/deactivate mechanics
- Add/update spare parts
- Adjust spare part stock (add/remove + reason)
- View reports

SERVICE_ADVISOR

- Register customer
- Register vehicle
- Create service booking
- Generate invoice + payment
- Print invoice (console output)

Authentication can be simple: username + role input (not full security).
Role gating: show/hide menu actions based on role.

B) Customer & Vehicle Management

Customer must have:

- customerId (auto)
- name
- phone (unique)
- createdAt

Operations:

- Create customer
- Search customer by phone

Vehicle must have:

- vehicleId (auto)
- customerId FK
- vehicleNumber (unique)
- brand
- model
- createdAt

Operations:

- Register vehicle
- Search vehicle by vehicleNumber

C) Mechanics

Each mechanic must have:

- mechanicId (auto)
- name
- specialization
- active (true/false)
- createdAt

Operations:

- Add mechanic
- Deactivate mechanic
- List mechanics

D) Spare Parts (Inventory Equivalent)

Each spare part must have:

- partId (auto)
- partCode (unique)
- name
- unitPrice
- availableQuantity
- reorderLevel
- active
- createdAt

Operations:

- Create spare part
- Update spare part price
- Adjust stock (admin only)
- Low stock report (availableQuantity <= reorderLevel)

Audit requirement: every stock change must insert a row in stock_adjustments with deltaQty and reason.

E) Service Booking (Core Workflow)

A booking includes:

- bookingId
- vehicleId
- mechanicId
- serviceDate
- status (CREATED, IN_PROGRESS, COMPLETED, CANCELLED)
- createdAt

Rules:

- Cannot assign inactive mechanic
- Cannot double-book same mechanic on same date
- Status transitions must be logical

F) Invoice Generation (Core Transaction)

An invoice includes:

- invoiceId
- bookingId FK
- laborCharge
- partsTotal
- totalAmount
- status (CREATED, PAID)
- createdAt

Invoice items:

- partId
- quantity
- unitPriceAtSale
- lineTotal

Rules:

- Cannot use more parts than available stock
- Payment marks invoice as PAID
- Must be transactional: if any step fails → rollback everything

Design rule (to avoid inconsistent implementations):

✅ Stock is decremented only after payment is SUCCESS.
If payment fails → invoice remains CREATED and stock remains unchanged.

G) Payments

Payment details:

- paymentId
- invoiceId FK
- mode (CASH, CARD, UPI)
- amount
- status (SUCCESS, FAILED)
- paidAt

Rules:

- Amount must equal totalAmount (v1)

H) Reporting Requirements (SQL-heavy)

Implement as menu options:

1. Daily Revenue Summary
    - Date → total invoices, total revenue, top 3 spare parts used

2. Mechanic Performance Report
    - mechanic-wise completed jobs
    - revenue generated
3. Low Stock Parts Report
4. Service History by Vehicle Number

Reports must use SQL joins/aggregations.

---

5) Layered Architecture (Required)

Use this structure (or very close):

- presentation/controller (console menus, input)
- service (business rules, validation, transactions)
- dao (JDBC code only)
- model (POJOs/entities)
- exception (custom exceptions)
- util (DB connection, validators, helpers)
- config/resources (db.properties, log4j2.xml)

Rule: UI layer never directly calls DAO.

---

6) Exception Handling Rules (Non-negotiable)

Create custom exceptions such as:

- ValidationException
- EntityNotFoundException
- InsufficientStockException
- DoubleBookingException
- DatabaseOperationException

Guidelines:

- Validate early in service layer
- Catch SQLExceptions in DAO and wrap into DatabaseOperationException
- Log errors with stack traces at ERROR level

- Show user-friendly message in console (no stack trace in UI)

---

7) JUnit Testing Requirements (Must Have)

You must include at least 12 meaningful tests, including these minimum required tests:

1. shouldRejectDuplicateVehicleNumber()
2. shouldRejectInactiveMechanicAssignment()
3. shouldPreventDoubleBookingForSameDate()
4. shouldRejectNegativeLaborCharge()
5. shouldRejectInvoiceWithZeroItems()
6. shouldThrowInsufficientStockWhenPartQtyExceedsAvailable()
7. shouldComputeInvoiceTotalCorrectly()
8. shouldFailPaymentWhenAmountMismatch()
9. shouldRollbackWhenInvoiceFails()
10. shouldDeactivateMechanicAndPreventBooking()
11. shouldNormalizePhoneDuringCustomerRegistration()
12. shouldReturnMechanicPerformanceAggregatedCorrectly()

Clear naming example:

- shouldThrowInsufficientStockWhenPartQtyExceedsAvailable()

---

8) Git Requirements

Repository must show:

- main branch protected in spirit (don’t commit directly)
- Feature branches like:
    - feature/booking-flow
    - feature/invoice
    - feature/report
- At least 20 commits with meaningful messages

---

9) Submission Deliverables

You must submit:

- Source code (Git repo)
- README.md
- schema.sql
- db.properties template (no passwords committed)
- log4j2.xml or log4j2.properties
- JUnit test output (screenshots or console output)

---

10) Evaluation Rubric

- Architecture & separation of concerns: 25%
- SQL schema + query quality + JDBC correctness: 20%
- Invoice transaction correctness + rollback safety: 15%

- Testing quality (JUnit): 15%
- Logging & exception handling: 10%
- Git hygiene + README quality: 10%
- Code cleanliness (naming, SRP, duplication): 5%

Starter Kit

None

```
|
  autocare-wms/
  ├─ README.md
  |  └─ HINT: Setup, DB steps, how to run, sample menu flows, and design decisions (esp. transaction timing + stock update).
  |
  ├─ schema.sql
  |  └─ HINT: Tables + constraints + seed data (8+ spare parts, 5+ mechanics). Normalize and enforce FK/UNIQUE/NOT NULL.
  |
  ├─ pom.xml                        (or build.gradle)
  |  └─ HINT: Add dependencies: JDBC driver (MySQL/Postgres), Log4j2, JUnit5 + Surefire plugin.
  |
  ├─ src/
  |  ├─ main/
  |  |  ├─ java/
  |  |  |  └─ com/
  |  |  |     └─ autocare/
  |  |  |        └─ wms/
  |  |  |           ├─ App.java
  |  |  |           |  └─ HINT: Entry point. Bootstraps config, shows main menu, routes to controllers, handles global errors.
  |  |  |           |
  |  |  |           ├─ config/
  |  |  |           |  ├─ AppConfig.java
  |  |  |           |  |  └─ HINT: Central place to wire services/daos (simple manual DI). Keep object creation here.
  |  |  |           |  └─ DbConfig.java
  |  |  |           |     └─ HINT: Read db.properties and expose DB URL/username/driver. No hardcoded credentials.
  |  |  |           |
  |  |  |           ├─ controller/
  |  |  |           |  ├─ CustomerController.java
  |  |  |           |  |  └─ HINT: Register/search customers by phone; used before vehicle registration and booking.
  |  |  |           |  ├─ VehicleController.java
  |  |  |           |  |  └─ HINT: Register/search vehicles by vehicle number; link vehicle to customer.
  |  |  |           |  ├─ MechanicController.java
  |  |  |           |  |  └─ HINT: Admin menu for mechanics (add/update/deactivate/list). Calls MechanicService.
  |  |  |           |  ├─ PartController.java
  |  |  |           |  |  └─ HINT: Admin menu for spare parts (CRUD, stock adjust, low stock view). Calls PartService.
  |  |  |           |  ├─ BookingController.java
  |  |  |           |  |  └─ HINT: Advisor menu to create bookings, assign mechanic, update booking status, view bookings.
  |  |  |           |  ├─ InvoiceController.java
  |  |  |           |  |  └─ HINT: Advisor menu to generate invoice: add parts + labor + payment + print invoice output.
  |  |  |           |  └─ ReportController.java
  |  |  |           |     └─ HINT: Report menus: daily revenue, mechanic performance, low stock, vehicle service history.
  |  |  |           |
  |  |  |           ├─ dao/
  |  |  |           |  ├─ CustomerDao.java
  |  |  |           |  |  └─ HINT: Create/find/search customers. Phone should be UNIQUE.
  |  |  |           |  ├─ VehicleDao.java
  |  |  |           |  |  └─ HINT: Create/find vehicles. Vehicle number should be UNIQUE.
  |  |  |           |  ├─ MechanicDao.java
  |  |  |           |  |  └─ HINT: CRUD mechanics + list active mechanics. No business logic here.
  |  |  |           |  ├─ PartDao.java
  |  |  |           |  |  └─ HINT: CRUD spare parts + find by partCode + search by name/code.
  |  |  |           |  ├─ PartInventoryDao.java
  |  |  |           |  |  └─ HINT: Stock read/update for parts. Must support safe updates (no negative stock).
  |  |  |           |  ├─ BookingDao.java
  |  |  |           |  |  └─ HINT: Create booking + update status + find booking by id/vehicle/date.
  |  |  |           |  ├─ InvoiceDao.java
  |  |  |           |  |  └─ HINT: Insert invoice + items + update totals/status + fetch invoice for printing.
  |  |  |           |  ├─ PaymentDao.java
  |  |  |           |  |  └─ HINT: Insert payment + fetch by invoiceId. One payment per invoice in v1.
  |  |  |           |  ├─ StockAdjustmentDao.java
  |  |  |           |  |  └─ HINT: Insert part stock adjustments (delta + reason). Optional: query history (bonus).
  |  |  |           |  ├─ ReportDao.java
  |  |  |           |  |  └─ HINT: Only SQL-heavy report queries (joins/aggregates). No printing; return DTO rows.
```

```
|  |  |  |           |  └─ impl/
|  |  |  |           |     ├─ JdbcCustomerDao.java
|  |  |  |           |     |  └─ HINT: PreparedStatements only; wrap SQLException → DatabaseOperationException.
|  |  |  |           |     ├─ JdbcVehicleDao.java
|  |  |  |           |     |  └─ HINT: Create vehicle + find by vehicle number; enforce customer FK usage.
|  |  |  |           |     ├─ JdbcMechanicDao.java
|  |  |  |           |     |  └─ HINT: CRUD mechanics, list active mechanics, deactivate via soft delete.
|  |  |  |           |     ├─ JdbcPartDao.java
|  |  |  |           |     |  └─ HINT: CRUD spare parts; search by name/code; return active parts by default.
|  |  |  |           |     ├─ JdbcPartInventoryDao.java
|  |  |  |           |     |  └─ HINT: Implement safe stock updates; use conditions to prevent negative quantity.
|  |  |  |           |     ├─ JdbcBookingDao.java
|  |  |  |           |     |  └─ HINT: Insert booking + update status; check conflicts through query methods.
|  |  |  |           |     ├─ JdbcInvoiceDao.java
|  |  |  |           |     |  └─ HINT: Insert invoice header + batch insert invoice_items + update totals/status.
|  |  |  |           |     ├─ JdbcPaymentDao.java
|  |  |  |           |     |  └─ HINT: Insert payment record; paidAt uses LocalDateTime.
|  |  |  |           |     ├─ JdbcStockAdjustmentDao.java
|  |  |  |           |     |  └─ HINT: Insert adjustment record per admin stock change (delta + reason).
|  |  |  |           |     └─ JdbcReportDao.java
|  |  |  |           |        └─ HINT: Implement reporting queries (GROUP BY, SUM, COUNT, TOP items).
|  |  |  |           |
|  |  |  |           ├─ exception/
|  |  |  |           |  ├─ ValidationException.java
|  |  |  |           |  |  └─ HINT: Throw when input/business rule fails (invalid price/qty/date/phone).
|  |  |  |           |  ├─ EntityNotFoundException.java
|  |  |  |           |  |  └─ HINT: Throw when DB entity missing (customer/vehicle/booking/part not found).
|  |  |  |           |  ├─ InsufficientStockException.java
|  |  |  |           |  |  └─ HINT: Throw when invoice uses more parts than available quantity.
|  |  |  |           |  ├─ DoubleBookingException.java
|  |  |  |           |  |  └─ HINT: Throw when mechanic is already booked for same serviceDate.
|  |  |  |           |  └─ DatabaseOperationException.java
|  |  |  |           |     └─ HINT: Wrap SQLExceptions here; UI prints friendly message; logs keep stack traces.
|  |  |  |           |
|  |  |  |           ├─ model/
|  |  |  |           |  ├─ Customer.java
|  |  |  |           |  |  └─ HINT: POJO for customers table (id, name, phone, createdAt).
|  |  |  |           |  ├─ Vehicle.java
|  |  |  |           |  |  └─ HINT: POJO for vehicles table (vehicleNumber unique, link to customer).
|  |  |  |           |  ├─ Mechanic.java
|  |  |  |           |  |  └─ HINT: POJO for mechanics (active true/false).
|  |  |  |           |  ├─ SparePart.java
|  |  |  |           |  |  └─ HINT: POJO for parts (partCode unique, price, active).
|  |  |  |           |  ├─ PartInventory.java
|  |  |  |           |  |  └─ HINT: POJO for part stock (availableQuantity, reorderLevel).
|  |  |  |           |  ├─ Booking.java
|  |  |  |           |  |  └─ HINT: POJO for booking header (serviceDate, status).
|  |  |  |           |  ├─ Invoice.java
|  |  |  |           |  |  └─ HINT: POJO for invoice header + list of InvoiceItem; status uses enum (CREATED/PAID).
|  |  |  |           |  ├─ InvoiceItem.java
|  |  |  |           |  |  └─ HINT: POJO for invoice line items (partId, qty, unitPriceAtSale, lineTotal).
|  |  |  |           |  ├─ Payment.java
|  |  |  |           |  |  └─ HINT: POJO for payment (mode, amount, status, paidAt).
|  |  |  |           |  └─ report/                        (DTOs for reports)
|  |  |  |           |     ├─ DailyRevenueRow.java
|  |  |  |           |     |  └─ HINT: DTO row for daily revenue summary results.
|  |  |  |           |     ├─ TopPartRow.java
|  |  |  |           |     |  └─ HINT: DTO row for top-used spare parts in a day.
|  |  |  |           |     ├─ MechanicPerformanceRow.java
|  |  |  |           |     |  └─ HINT: DTO row for mechanic-wise jobs + revenue.
|  |  |  |           |     ├─ LowStockPartRow.java
|  |  |  |           |     |  └─ HINT: DTO row for low-stock listing output.
|  |  |  |           |     └─ VehicleServiceHistoryRow.java
|  |  |  |           |        └─ HINT: DTO row for vehicle service history (date, booking status, invoice totals).
|  |  |  |           |
|  |  |  |           ├─ service/
|  |  |  |           |  ├─ CustomerService.java
|  |  |  |           |  |  └─ HINT: Validate + normalize phone; create/find customers; getOrCreate by phone.
|  |  |  |           |  ├─ VehicleService.java
|  |  |  |           |  |  └─ HINT: Register vehicles; enforce unique vehicleNumber; ensure customer exists.
|  |  |  |           |  ├─ MechanicService.java
|  |  |  |           |  |  └─ HINT: Admin validations; deactivate mechanic; list active mechanics.
```

```
|  |  |           |  ├─ PartService.java
|  |  |           |  |  └─ HINT: Validate partCode/price; create/update/deactivate parts; search parts.
|  |  |           |  ├─ PartInventoryService.java
|  |  |           |  |  └─ HINT: Admin stock adjust + audit insert. Ensure no negative stock.
|  |  |           |  ├─ BookingService.java
|  |  |           |  |  └─ HINT: Create booking; enforce mechanic active + no double-booking; manage status transitions.
|  |  |           |  ├─ InvoiceService.java
|  |  |           |  |  └─ HINT: Core transaction: validate parts stock, compute totals, insert invoice+items, payment,
|  |  |           |  |           decrement stock on payment SUCCESS, commit; rollback on any failure.
|  |  |           |  └─ ReportService.java
|  |  |           |     └─ HINT: Validate date/range; call ReportDao; return DTO rows for printing.
|  |  |           |
|  |  |           ├─ util/
|  |  |           |  ├─ DbConnectionFactory.java
|  |  |           |  |  └─ HINT: Reads db.properties and returns Connection. Used by DAO/service transaction boundaries.
|  |  |           |  ├─ InputUtil.java
|  |  |           |  |  └─ HINT: Safe console input parsing for int/date/string; avoid crashing on invalid input.
|  |  |           |  ├─ ValidationUtil.java
|  |  |           |  |  └─ HINT: Validators for phone normalization, vehicle number format, partCode, qty > 0, etc.
|  |  |           |  ├─ MoneyUtil.java
|  |  |           |  |  └─ HINT: BigDecimal parsing/formatting (scale=2). No doubles in calculations.
|  |  |           |  └─ DateUtil.java
|  |  |           |     └─ HINT: Parse serviceDate and date ranges consistently (ISO format recommended).
|  |  |           |
|  |  |           └─ constants/
|  |  |              ├─ Role.java
|  |  |              |  └─ HINT: Enum for ADMIN/SERVICE_ADVISOR; used in simple login and menu gating.
|  |  |              ├─ BookingStatus.java
|  |  |              |  └─ HINT: CREATED/IN_PROGRESS/COMPLETED/CANCELLED stored as text in DB.
|  |  |              ├─ InvoiceStatus.java
|  |  |              |  └─ HINT: CREATED/PAID stored as text in DB.
|  |  |              ├─ PaymentMode.java
|  |  |              |  └─ HINT: CASH/CARD/UPI input maps to enum.
|  |  |              └─ PaymentStatus.java
|  |  |                 └─ HINT: SUCCESS/FAILED stored as text in DB.
|  |  |
|  |  └─ resources/
|  |     ├─ db.properties
|  |     |  └─ HINT: Only DB URL/user; commit a template (no password). Mention in README.
|  |     └─ log4j2.xml
|  |        └─ HINT: Console + file appenders. INFO for flow, WARN for validations, ERROR for stack traces.
|  |
|  └─ test/
|     └─ java/
|        └─ com/autocare/wms/service/
|           ├─ BookingServiceTest.java
|           |  └─ HINT: Test double-booking + inactive mechanic assignment + status transition rules.
|           ├─ InvoiceServiceTest.java
|           |  └─ HINT: Test stock checks, totals, payment mismatch, and rollback behavior (mock/fake dao is ok).
|           ├─ PartServiceTest.java
|           |  └─ HINT: Test part validations + uniqueness of partCode.
|           └─ ValidationUtilTest.java
|              └─ HINT: Test edge cases for phone normalization, vehicle number, part code formats.
```

Minimal bootstrap classes (copy/paste)

Java

```java
package com.neomart.sist;

import com.neomart.sist.controller.OrderController;
import com.neomart.sist.controller.ProductController;
import com.neomart.sist.controller.ReportController;
import com.neomart.sist.util.InputUtil;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class App {
    private static final Logger log = LogManager.getLogger(App.class);

    public static void main(String[] args) {
        log.info("NeoMart S.I.S.T. started");

        ProductController productController = new ProductController();
        OrderController orderController = new OrderController();
        ReportController reportController = new ReportController();

        while (true) {
            System.out.println("\n=== NeoMart S.I.S.T. ===");
            System.out.println("1. Products");
            System.out.println("2. Orders");
            System.out.println("3. Reports");
            System.out.println("0. Exit");

            int choice = InputUtil.readInt("Choose: ");
            switch (choice) {
                case 1 -> productController.menu();
                case 2 -> orderController.menu();
                case 3 -> reportController.menu();
                case 0 -> {
                    log.info("NeoMart S.I.S.T. stopped");
                    System.out.println("Bye!");
                    return;
                }
                default -> System.out.println("Invalid option.");
            }
        }
    }
}
```

`DbConnectionFactory.java` (single place for connections)

Java

```java
package com.autocare.wms.util;

import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;

public final class DbConnectionFactory {
```

```
    private static final Properties props = new Properties();

    static {
        try (InputStream in =
DbConnectionFactory.class.getClassLoader().getResourceAsStream("db.propertie
s")) {
            if (in == null) throw new IllegalStateException("db.properties
not found in resources/");
            props.load(in);
        } catch (Exception e) {
            throw new ExceptionInInitializerError("Failed to load
db.properties: " + e.getMessage());
        }
    }

    private DbConnectionFactory() {}

    public static Connection getConnection() {
        try {
            return DriverManager.getConnection(
                    props.getProperty("db.url"),
                    props.getProperty("db.username"),
                    props.getProperty("db.password")
            );
        } catch (Exception e) {
            throw new RuntimeException("DB connection failed: " +
e.getMessage(), e);
        }
    }
}
```

DB Schema

None

```
-- ========================================================
-- AutoCare WMS - Database Schema (MySQL 8+)
-- ========================================================

-- Create DB
CREATE DATABASE IF NOT EXISTS autocare_wms;
USE autocare_wms;

-- ========================================================
-- Customers
-- ========================================================
CREATE TABLE customers (
  customer_id  BIGINT PRIMARY KEY AUTO_INCREMENT,
  name         VARCHAR(120) NOT NULL,
  phone        VARCHAR(20)  NOT NULL UNIQUE,
```

```
  created_at   DATETIME     NOT NULL DEFAULT CURRENT_TIMESTAMP
);

-- ============================================================
-- Vehicles (each vehicle belongs to a customer)
-- ============================================================
CREATE TABLE vehicles (
  vehicle_id      BIGINT PRIMARY KEY AUTO_INCREMENT,
  customer_id     BIGINT      NOT NULL,
  vehicle_number  VARCHAR(30) NOT NULL UNIQUE,
  brand           VARCHAR(60) NOT NULL,
  model           VARCHAR(60) NOT NULL,
  created_at      DATETIME    NOT NULL DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT fk_vehicle_customer
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
    ON DELETE RESTRICT
);

-- ============================================================
-- Mechanics
-- ============================================================
CREATE TABLE mechanics (
  mechanic_id     BIGINT PRIMARY KEY AUTO_INCREMENT,
  name            VARCHAR(120) NOT NULL,
  specialization  VARCHAR(80)  NOT NULL,
  active          BOOLEAN      NOT NULL DEFAULT TRUE,
  created_at      DATETIME     NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at      DATETIME     NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

-- ============================================================
-- Spare Parts (master)
-- ============================================================
CREATE TABLE spare_parts (
  part_id      BIGINT PRIMARY KEY AUTO_INCREMENT,
  part_code    VARCHAR(50)  NOT NULL UNIQUE,
  name         VARCHAR(120) NOT NULL,
  unit_price   DECIMAL(10,2) NOT NULL,
  active       BOOLEAN       NOT NULL DEFAULT TRUE,
  created_at   DATETIME      NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at   DATETIME      NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

-- ============================================================
-- Part Inventory (1 row per part)
-- ============================================================
CREATE TABLE part_inventory (
  part_id            BIGINT PRIMARY KEY,
  available_quantity INT NOT NULL,
  reorder_level      INT NOT NULL DEFAULT 5,
  updated_at         DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  CONSTRAINT fk_inventory_part
    FOREIGN KEY (part_id) REFERENCES spare_parts(part_id)
    ON DELETE RESTRICT
);

-- ============================================================
-- Service Bookings
-- ============================================================
CREATE TABLE bookings (
  booking_id   BIGINT PRIMARY KEY AUTO_INCREMENT,
  vehicle_id   BIGINT NOT NULL,
  mechanic_id  BIGINT NOT NULL,
  service_date DATE   NOT NULL,
  status       VARCHAR(20) NOT NULL, -- CREATED, IN_PROGRESS, COMPLETED, CANCELLED
  created_at   DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,

  CONSTRAINT fk_booking_vehicle
    FOREIGN KEY (vehicle_id) REFERENCES vehicles(vehicle_id)
    ON DELETE RESTRICT,
```

```
  CONSTRAINT fk_booking_mechanic
    FOREIGN KEY (mechanic_id) REFERENCES mechanics(mechanic_id)
    ON DELETE RESTRICT,

  -- Prevent double-booking same mechanic on same date
  CONSTRAINT uq_mechanic_date UNIQUE (mechanic_id, service_date)
);

-- ============================================================
-- Invoices (1 invoice per booking in v1)
-- ============================================================
CREATE TABLE invoices (
  invoice_id     BIGINT PRIMARY KEY AUTO_INCREMENT,
  booking_id     BIGINT NOT NULL UNIQUE,
  labor_charge   DECIMAL(10,2) NOT NULL,
  parts_total    DECIMAL(12,2) NOT NULL,
  total_amount   DECIMAL(12,2) NOT NULL,
  status         VARCHAR(20) NOT NULL, -- CREATED, PAID
  created_at     DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,

  CONSTRAINT fk_invoice_booking
    FOREIGN KEY (booking_id) REFERENCES bookings(booking_id)
    ON DELETE RESTRICT
);

-- ============================================================
-- Invoice Items (spare parts used)
-- ============================================================
CREATE TABLE invoice_items (
  invoice_item_id    BIGINT PRIMARY KEY AUTO_INCREMENT,
  invoice_id         BIGINT NOT NULL,
  part_id            BIGINT NOT NULL,
  quantity           INT NOT NULL,
  unit_price_at_sale DECIMAL(10,2) NOT NULL,
  line_total         DECIMAL(12,2) NOT NULL,

  CONSTRAINT fk_items_invoice
    FOREIGN KEY (invoice_id) REFERENCES invoices(invoice_id)
    ON DELETE CASCADE,

  CONSTRAINT fk_items_part
    FOREIGN KEY (part_id) REFERENCES spare_parts(part_id)
    ON DELETE RESTRICT
);

-- ============================================================
-- Payments (1 payment per invoice in v1)
-- ============================================================
CREATE TABLE payments (
  payment_id   BIGINT PRIMARY KEY AUTO_INCREMENT,
  invoice_id   BIGINT NOT NULL UNIQUE,
  mode         VARCHAR(20) NOT NULL, -- CASH, CARD, UPI
  amount       DECIMAL(12,2) NOT NULL,
  status       VARCHAR(20) NOT NULL, -- SUCCESS, FAILED
  paid_at      DATETIME NULL,

  CONSTRAINT fk_payment_invoice
    FOREIGN KEY (invoice_id) REFERENCES invoices(invoice_id)
    ON DELETE CASCADE
);

-- ============================================================
-- Stock adjustment audit (admin stock changes with reason)
-- ============================================================
CREATE TABLE stock_adjustments (
  adjustment_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  part_id       BIGINT NOT NULL,
  delta_qty     INT NOT NULL, -- + for add, - for reduce
  reason        VARCHAR(200) NOT NULL,
  created_at    DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
```

```
  CONSTRAINT fk_adj_part
    FOREIGN KEY (part_id) REFERENCES spare_parts(part_id)
    ON DELETE RESTRICT
);

-- ============================================================
-- Optional CHECK constraints (MySQL 8+ enforces CHECK)
-- ============================================================
ALTER TABLE spare_parts
  ADD CONSTRAINT chk_part_unit_price CHECK (unit_price > 0);

ALTER TABLE part_inventory
  ADD CONSTRAINT chk_inv_qty CHECK (available_quantity >= 0),
  ADD CONSTRAINT chk_inv_reorder CHECK (reorder_level >= 0);

ALTER TABLE invoice_items
  ADD CONSTRAINT chk_item_qty CHECK (quantity > 0),
  ADD CONSTRAINT chk_item_line_total CHECK (line_total >= 0);

ALTER TABLE invoices
  ADD CONSTRAINT chk_invoice_labor CHECK (labor_charge >= 0),
  ADD CONSTRAINT chk_invoice_total CHECK (total_amount >= 0);

ALTER TABLE stock_adjustments
  ADD CONSTRAINT chk_adj_delta CHECK (delta_qty <> 0);

-- ============================================================
-- Seed Data (5 mechanics, 8 spare parts + inventory)
-- ============================================================
INSERT INTO mechanics (name, specialization, active) VALUES
('Ravi Patil', 'Engine', TRUE),
('Neha Sharma', 'Brakes', TRUE),
('Amit Verma', 'Electrical', TRUE),
('Sana Khan', 'AC', TRUE),
('John Dsouza', 'General Service', TRUE);

INSERT INTO spare_parts (part_code, name, unit_price, active) VALUES
('PART-1001', 'Engine Oil 1L', 450.00, TRUE),
('PART-1002', 'Oil Filter', 180.00, TRUE),
('PART-1003', 'Air Filter', 220.00, TRUE),
('PART-1004', 'Brake Pads Set', 1200.00, TRUE),
('PART-1005', 'Spark Plug', 320.00, TRUE),
('PART-1006', 'Wiper Blade', 300.00, TRUE),
('PART-1007', 'Coolant 1L', 280.00, TRUE),
('PART-1008', 'Headlight Bulb', 150.00, TRUE);

INSERT INTO part_inventory (part_id, available_quantity, reorder_level)
SELECT part_id, 30, 5 FROM spare_parts;

-- Optional seed customers + vehicles (helps demo flows)
INSERT INTO customers (name, phone) VALUES
('Karan Mehta', '9876543210'),
('Anita Joshi', '9123456780');

INSERT INTO vehicles (customer_id, vehicle_number, brand, model) VALUES
(1, 'MH12AB1234', 'Honda', 'City'),
(2, 'RJ19CD5678', 'Hyundai', 'i20');
```

Sure — here's the **Sample Menu Flow (console UX)** for **Project 2: AutoCare WMS (S.B.W.M.S.)**, written in the same style/structure as Project 1 & Project 3.

Sample menu flow (console UX) — Project 2 (AutoCare WMS)

Main Menu

```
=== AutoCare WMS ===

1. Customers & Vehicles

2. Bookings

3. Invoices & Payments

4. Parts & Stock (Admin)

5. Reports

0. Exit
```

> (If you prefer fewer top-level menus, you can merge “Customers & Vehicles” into “Bookings”, but the flow below remains the same.)

Customers & Vehicles Menu

```
--- Customers & Vehicles ---

1. Register customer

2. Search customer by phone

3. Register vehicle (link to customer)

4. Search vehicle by vehicle number

0. Back
```

Recommended flow for “Register vehicle”:

1. Ask customer phone → find customer (or create if new)
2. Enter vehicle number, brand, model

3. Validate unique vehicle number
4. Save vehicle linked to customer

Bookings Menu

```
--- Bookings ---
1. Create service booking (Service Advisor)
2. Cancel booking
3. Mark booking status (IN_PROGRESS / COMPLETED)
4. Find booking by ID
5. List bookings by date
0. Back
```

Recommended flow for "Create service booking":

1. Ask vehicle number → find vehicle (or register first)
2. Show active mechanics list
3. Choose mechanic + enter service date
4. Validate:
    - mechanic is active
    - mechanic is NOT already booked for that date (double booking rule)
5. Create booking with status CREATED
6. Print booking confirmation (bookingId)

Invoices & Payments Menu (Core Workflow)

```
--- Invoices & Payments ---

1. Generate invoice for booking

2. Add spare part item (during invoice creation)

3. Checkout + payment

4. Find invoice by ID

5. Print invoice receipt (by invoice ID)

0. Back
```

Recommended flow for “Generate invoice + payment”:

1. Ask bookingId
2. Validate booking exists AND status is COMPLETED (recommended rule)
    - If you allow billing before completion, document your decision clearly in README
3. Enter labor charge
4. Add spare parts in a loop: partCode + qty
5. Validate for each part:
    - part is active
    - qty > 0
    - qty <= available stock
6. Show invoice preview:
    - labor charge
    - parts total
    - total amount
7. Select payment mode + confirm

8. Execute invoice transactionally:
    - insert invoice header (CREATED)
    - insert invoice_items
    - insert payment (SUCCESS/FAILED)
    - if SUCCESS → decrement part_inventory quantities
    - update invoice status to PAID
    - commit
    - if any step fails → rollback everything

Parts & Stock Menu (Admin)

```
--- Parts & Stock (Admin) ---
1. Add spare part
2. Update spare part (name/price)
3. Deactivate spare part
4. Adjust stock (+/- with reason)
5. Search spare part (name/partCode)
6. View part by partCode
0. Back
```

Recommended flow for “Adjust stock”:

1. Ask partCode
2. Ask delta quantity (+ for add, - for reduce)
3. Ask reason (mandatory)

4. Validate: resulting stock cannot be negative
5. Update inventory + insert stock_adjustments record
6. Print updated stock quantity

Reports Menu (SQL-heavy)

```
--- Reports ---
1. Daily revenue summary (date)
2. Mechanic performance report (date range)
3. Low stock parts report
4. Service history by vehicle number
0. Back
```

Report details (expected output):

- Daily revenue: total invoices, revenue, top 3 parts used
- Mechanic performance: mechanic-wise completed jobs + revenue
- Low stock: parts where available_quantity <= reorder_level
- Service history: bookings + invoices/payments for a vehicle