**Student Management System**

**📍 Step 1: Set Up MySQL Database**

CREATE DATABASE StudentDB;

CREATE TABLE students (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100) NOT NULL,

age INT NOT NULL,

grade FLOAT NOT NULL);

**📍 Step 2: Set Up Java Project in IntelliJ**

**✅ 2. Create a Package and Java Class**

**Package name :-** com.student.management

**Java class name :-** StudentManagementApp

**📍 Step 3: JDBC Connection Setup**

1. **Inside DatabaseConnection.java (Class)**

private static final String URL = "jdbc:mysql://localhost:3306/StudentDB";

private static final String USER = "root";

private static final String PASSWORD = "your\_password";

• **URL** → Points to **StudentDB** running on **localhost** (default MySQL port: 3306).

• **USER & PASSWORD** → Replace with your MySQL **username and password**.

Class.forName("com.mysql.cj.jdbc.Driver");

• Ensures the **MySQL JDBC Driver** is loaded before making a connection.

• **Important:** If this fails, it means the JDBC JAR file is not added properly.

conn = DriverManager.getConnection(URL, USER, PASSWORD);

System.out.println("✅ Connected to Database Successfully!");

• **DriverManager.getConnection()** → Connects to **MySQL Database**.

• If successful, **prints a confirmation message**.

**📍 Step 4: Create the Student Class**

This will be our **Java Bean (POJO class)** to store student details.

**🔹 1. Create Student.java**

Inside the **com.student.management** package, create a new file:

Student.java

Attributes (Using Encapsulation)

• **Why private?**

• **Encapsulation** ensures data protection.

• Direct access is **not allowed** (modification via **getters/setters** only).

**🔹 Interview Questions on OOP (Java Basics)**

**1. Why do we use getters and setters?**

👉 Getters and setters provide **controlled access** to private fields, ensuring **data encapsulation**.

**2. What is Encapsulation in Java?**

👉 Encapsulation is **hiding data** inside a class and restricting access using **getters/setters**.

**3. Why do we override toString()?**

👉 The default toString() method prints **hashcode** of the object.

Overriding it gives **meaningful output**.

**Without overriding toString()**

👉 Output: com.student.management.Student@5acf9800

**With toString() overridden**

👉 Output: Student{id=1, name='Tejas', age=21, grade='A'}

**📍 Step 5: Create StudentDAO for CRUD Operations**

Implement **CRUD operations** (Create, Read, Update, Delete) using **JDBC**.

**🔹 1. Create StudentDAO.java**

Inside the **com.student.management** package, create a new file:

StudentDAO.java

**🔹 Explanation of CRUD Operations**

**1️. Constructor for DB Connection**

private Connection connection;

public StudentDAO(Connection connection) {

this.connection = connection;

}

**Why?**

• We pass the **JDBC connection** from DBConnection.java (created earlier).

• This prevents multiple unnecessary connections.

**2️. CREATE (Insert a Student)**

•? **placeholders** prevent SQL injection.

• **PreparedStatement** improves security and performance.

**3️. READ (Fetch All Students)**

• **Retrieves all records** from the students table.

• **Stores them in a List<Student>** for easy access.

**4️. UPDATE (Modify Student Details)**

**Why check rowsAffected?**

• If 0, **no matching student** exists.

• If >0, **update successful**.

**5️. DELETE (Remove Student Record)**

• Ensures **safe deletion** of a student by **ID**.

**🔹 Interview Questions (JDBC & CRUD)**

1. **What is the difference between Statement and PreparedStatement?**

**Feature** **Statement PreparedStatement**

**Query Execution** Compiled each time Precompiled once, reused

**Performance** Slower Faster

**Security** Prone to SQL Injection Prevents SQL Injection

1. **What is SQL Injection? How does PreparedStatement prevent it?**

**SQL Injection** is a hacking technique where attackers manipulate SQL queries.

1. **What happens if you don’t close JDBC connections?**

• **Memory Leaks**: Too many open connections slow the system.

• **Database Overload**: The DB server may **reject new connections**.

• **Fix**: Always use try-with-resources to **automatically close** resources.

**📍 Step 6: Create Main.java (Test the CRUD Operations)**

create a **menu-driven program** in Main.java to interact with the StudentDAO methods.

**🔹 1. Create Main.java**

Inside the **com.student.management** package, create a new file:

Main.java

**🔹 Explanation**

1. **Establish Database Connection**

try (Connection connection = DBConnection.getConnection();

Scanner scanner = new Scanner(System.in)) {

• DBConnection.getConnection() fetches a database connection.

• try-with-resources ensures **auto-closing** of DB connection & scanner.

**2️. Menu-Driven System**

• **Loop runs continuously** until the user chooses Exit (5).

• **User input (choice) decides CRUD operation**.

**3️. Handling Student Addition**

• **scanner.nextLine() is used after nextInt()** to prevent **input skipping issues**.

• Creates a Student object and **calls addStudent()**.

**4️. Viewing All Students**

List<Student> students = studentDAO.getAllStudents();

if (students.isEmpty()) {

System.out.println("No students found.");

} else {

System.out.println("\n==== Student List ====");

for (Student s : students) {

System.out.println(s);

}

}

• Fetches student list using getAllStudents().

• **Checks if list is empty** before displaying.

**5️.Updating a Student**

System.out.print("Enter Student ID to update: ");

int updateId = scanner.nextInt();

scanner.nextLine();

System.out.print("Enter New Name: ");

String newName = scanner.nextLine();

System.out.print("Enter New Age: ");

int newAge = scanner.nextInt();

scanner.nextLine();

System.out.print("Enter New Grade: ");

String newGrade = scanner.nextLine();

Student updatedStudent = new Student(updateId, newName, newAge, newGrade);

studentDAO.updateStudent(updatedStudent);

Takes **ID of student** to update.

• Calls updateStudent() with modified details.

**6️. Deleting a Student**

System.out.print("Enter Student ID to delete: ");

int deleteId = scanner.nextInt();

studentDAO.deleteStudent(deleteId);

• Calls deleteStudent() method using **student ID**.

**7️. Exit the Program**

case 5:

System.out.println("Exiting Student Management System.");

return;

• **Terminates the loop & exits the program**.