**Spring JDBC**

**1. Introduction to Spring JDBC**

* Limitations of traditional JDBC
* Advantages of Spring JDBC
* Key features of Spring JDBC
* Overview of JdbcTemplate and Spring DataSource support

**Introduction to Spring JDBC**

**Spring JDBC** is a module in the Spring Framework that simplifies database operations by providing a higher-level abstraction over traditional JDBC (Java Database Connectivity). It streamlines interaction with relational databases, reducing boilerplate code and handling low-level details like resource management and exception handling. Spring JDBC is designed to make database access more efficient, maintainable, and less error-prone while retaining the flexibility of JDBC.

**1. Limitations of Traditional JDBC**

Traditional JDBC, while powerful, has several limitations that make it cumbersome for developers:

* **Boilerplate Code**: JDBC requires repetitive code for managing database connections, preparing statements, handling result sets, and closing resources. This increases development time and the risk of errors.
* **Resource Management**: Developers must manually manage database connections, statements, and result sets, ensuring they are properly closed to avoid resource leaks.
* **Exception Handling**: JDBC throws checked SQLException for most operations, forcing developers to write extensive try-catch blocks, which clutter code and make it harder to maintain.
* **Error Code Handling**: JDBC provides vendor-specific error codes, requiring developers to handle database-specific errors manually, reducing portability.
* **Lack of Abstraction**: JDBC operates at a low level, lacking built-in support for common tasks like mapping query results to Java objects or batch operations.
* **Transaction Management**: Managing transactions in JDBC requires manual handling of commit and rollback operations, which can be error-prone.

**2. Advantages of Spring JDBC**

Spring JDBC addresses the limitations of traditional JDBC and provides several advantages:

* **Reduced Boilerplate Code**: Spring JDBC eliminates repetitive code by providing utilities like JdbcTemplate, which handles resource management and simplifies query execution.
* **Simplified Exception Handling**: Spring JDBC wraps SQLException into unchecked DataAccessException hierarchies, making exception handling cleaner and more consistent across databases.
* **Resource Management**: Spring automatically manages database connections, statements, and result sets, ensuring resources are properly closed to prevent leaks.
* **Database Portability**: Spring abstracts vendor-specific error codes into a consistent exception hierarchy, improving portability across different databases.
* **Simplified Transaction Management**: Spring provides declarative and programmatic transaction management, reducing the need for manual transaction handling.
* **Support for Common Operations**: Spring JDBC supports common database operations like querying, updating, batch processing, and mapping results to Java objects with minimal code.
* **Integration with Spring Ecosystem**: Spring JDBC integrates seamlessly with other Spring modules (e.g., Spring ORM, Spring Transaction) and dependency injection, enabling modular and testable applications.

**3. Key Features of Spring JDBC**

Spring JDBC offers several features that make it a robust choice for database access:

* **JdbcTemplate**: A central class that simplifies database operations by providing methods for executing SQL queries, updates, and batch operations. It handles resource management and exception translation automatically.
* **NamedParameterJdbcTemplate**: An extension of JdbcTemplate that supports named parameters in SQL queries, improving readability and maintainability.
* **Exception Translation**: Converts vendor-specific SQLException into Spring’s DataAccessException hierarchy, providing consistent and meaningful error handling.
* **Connection Management**: Integrates with Spring’s DataSource to manage database connections efficiently, supporting connection pooling and configuration.
* **Row Mapping**: Provides utilities like RowMapper and ResultSetExtractor to map query results to Java objects, reducing manual result set processing.
* **Batch Processing**: Supports efficient batch operations for executing multiple SQL statements in a single database call, improving performance.
* **Transaction Support**: Offers programmatic and declarative transaction management, integrating with Spring’s transaction infrastructure.
* **Embedded Database Support**: Simplifies testing by providing support for embedded databases like H2, HSQL, and Derby.
* **Flexible Query Execution**: Supports prepared statements, callable statements, and dynamic SQL with parameter binding.

**4. Overview of JdbcTemplate and Spring DataSource Support**

**JdbcTemplate**

JdbcTemplate is the cornerstone of Spring JDBC, providing a simple and consistent API for database operations. It encapsulates low-level JDBC details, allowing developers to focus on writing SQL queries and processing results.

* **Key Methods**:
  + query(): Executes SELECT queries and maps results to Java objects using RowMapper or ResultSetExtractor.
  + update(): Executes INSERT, UPDATE, or DELETE statements.
  + execute(): Executes arbitrary SQL statements, including DDL (e.g., creating tables).
  + batchUpdate(): Performs batch operations for multiple SQL statements.
  + queryForObject(): Retrieves a single object from a query result.
* **Example**:

***import org.springframework.jdbc.core.JdbcTemplate;***

***import org.springframework.jdbc.datasource.DriverManagerDataSource;***

***public class JdbcTemplateExample {***

***public static void main(String[] args) {***

***// Configure DataSource***

***DriverManagerDataSource dataSource = new DriverManagerDataSource();***

***dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");***

***dataSource.setUrl("jdbc:mysql://localhost:3306/mydb");***

***dataSource.setUsername("root");***

***dataSource.setPassword("password");***

***// Initialize JdbcTemplate***

***JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);***

***// Execute a query***

***String sql = "SELECT name FROM employee WHERE id = ?";***

***String name = jdbcTemplate.queryForObject(sql, String.class, 1);***

***System.out.println("Employee Name: " + name);***

***}***

***}***

* **Benefits**:
  + Eliminates boilerplate code for connection management and result set handling.
  + Automatically closes resources (connections, statements, result sets).
  + Simplifies parameter binding and result mapping.

**Spring DataSource Support**

Spring JDBC relies on a DataSource to manage database connections. The DataSource interface provides a standard way to obtain connections, supporting various implementations like connection pools or embedded databases.

* **Common DataSource Implementations**:
  + **DriverManagerDataSource**: A simple implementation for basic use cases, creating a new connection for each request (not suitable for production due to lack of pooling).
  + **Apache Commons DBCP**: Provides connection pooling for improved performance in production environments.
  + **HikariCP**: A high-performance connection pool, widely used in modern applications.
  + **Embedded Databases**: Spring supports embedded databases like H2, HSQL, and Derby for testing purposes.
* **Configuration Example (Using Spring XML)**:

***<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">***

***<property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>***

***<property name="url" value="jdbc:mysql://localhost:3306/mydb"/>***

***<property name="username" value="root"/>***

***<property name="password" value="password"/>***

***</bean>***

***<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">***

***<property name="dataSource" ref="dataSource"/>***

***</bean>***

* **Benefits of DataSource Support**:
  + Centralized configuration of database connection details.
  + Integration with connection pools for efficient resource usage.
  + Support for dependency injection, allowing easy configuration in Spring applications.
  + Simplifies switching between different database environments (e.g., development, testing, production).

Spring JDBC simplifies database access by addressing the limitations of traditional JDBC, such as boilerplate code, resource management, and exception handling. Its key features, including JdbcTemplate, NamedParameterJdbcTemplate, and robust DataSource support, make it a powerful tool for interacting with relational databases. By reducing complexity and providing seamless integration with the Spring ecosystem, Spring JDBC enables developers to write cleaner, more maintainable, and scalable database code.

**2. Setting Up Spring JDBC Environment**

* Required dependencies (Maven/Gradle)
* Configuration using:
  + **XML-based**
  + **Java-based (AnnotationConfig)**
* Defining DataSource:
  + Using DriverManagerDataSource (basic)
  + Using HikariCP / Apache DBCP2 (production-grade)

**Setting Up Spring JDBC Environment**

To use **Spring JDBC** in a Java application, you need to set up the environment by including the necessary dependencies, configuring the Spring application context, and defining a DataSource for database connectivity. This section explains the required dependencies, configuration approaches (XML-based and Java-based), and how to define a DataSource using both basic and production-grade implementations.

**1. Required Dependencies (Maven/Gradle)**

To use Spring JDBC, you need to include the Spring JDBC module and a JDBC driver for your database. Optionally, you can include a connection pool library for production-grade applications.

**Maven Dependencies:** Add the following dependencies to your pom.xml:

***<dependencies>***

***<!-- Spring JDBC -->***

***<dependency>***

***<groupId>org.springframework</groupId>***

***<artifactId>spring-jdbc</artifactId>***

***<version>6.1.13</version> <!-- Use the latest version -->***

***</dependency>***

***<!-- JDBC Driver (e.g., MySQL) -->***

***<dependency>***

***<groupId>mysql</groupId>***

***<artifactId>mysql-connector-java</artifactId>***

***<version>8.0.33</version> <!-- Use the latest version -->***

***</dependency>***

***<!-- Optional: HikariCP for connection pooling -->***

***<dependency>***

***<groupId>com.zaxxer</groupId>***

***<artifactId>HikariCP</artifactId>***

***<version>5.1.0</version> <!-- Use the latest version -->***

***</dependency>***

***<!-- Optional: Apache DBCP2 for connection pooling -->***

***<dependency>***

***<groupId>org.apache.commons</groupId>***

***<artifactId>commons-dbcp2</artifactId>***

***<version>2.12.0</version> <!-- Use the latest version -->***

***</dependency>***

***</dependencies>***

**Notes**:

* Replace the MySQL driver with the appropriate driver for your database (e.g., org.postgresql:postgresql for PostgreSQL).
* The Spring JDBC dependency includes JdbcTemplate and other utilities.
* Connection pooling libraries (HikariCP or Apache DBCP2) are optional but recommended for production.

**2. Configuration Using XML-Based and Java-Based (AnnotationConfig)**

Spring JDBC can be configured using either **XML-based configuration** or **Java-based configuration** (using annotations). Both approaches define a DataSource and JdbcTemplate beans.

**XML-Based Configuration**

In XML-based configuration, you define beans in a Spring configuration file (e.g., applicationContext.xml).

**Example: XML Configuration**

***<?xml version="1.0" encoding="UTF-8"?>***

***<beans xmlns="http://www.springframework.org/schema/beans"***

***xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"***

***xsi:schemaLocation="http://www.springframework.org/schema/beans***

***http://www.springframework.org/schema/beans/spring-beans.xsd">***

***<!-- DataSource Configuration -->***

***<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">***

***<property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>***

***<property name="url" value="jdbc:mysql://localhost:3306/mydb"/>***

***<property name="username" value="root"/>***

***<property name="password" value="password"/>***

***</bean>***

***<!-- JdbcTemplate Configuration -->***

***<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">***

***<property name="dataSource" ref="dataSource"/>***

***</bean>***

***</beans>***

**Usage**:

* Load the XML configuration in your application:

***ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");***

***JdbcTemplate jdbcTemplate = context.getBean("jdbcTemplate", JdbcTemplate.class);***

**Notes**:

* The dataSource bean defines database connection details.
* The jdbcTemplate bean is wired with the dataSource for database operations.
* XML configuration is verbose but useful for legacy applications or when you prefer external configuration.

**Java-Based Configuration (AnnotationConfig)**

In Java-based configuration, you use Java classes with annotations like @Configuration and @Bean to define the Spring context.

**Example: Java Configuration**

***import org.springframework.context.annotation.Bean;***

***import org.springframework.context.annotation.Configuration;***

***import org.springframework.jdbc.core.JdbcTemplate;***

***import org.springframework.jdbc.datasource.DriverManagerDataSource;***

***import javax.sql.DataSource;***

***@Configuration***

***public class SpringConfig {***

***@Bean***

***public DataSource dataSource() {***

***DriverManagerDataSource dataSource = new DriverManagerDataSource();***

***dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");***

***dataSource.setUrl("jdbc:mysql://localhost:3306/mydb");***

***dataSource.setUsername("root");***

***dataSource.setPassword("password");***

***return dataSource;***

***}***

***@Bean***

***public JdbcTemplate jdbcTemplate(DataSource dataSource) {***

***return new JdbcTemplate(dataSource);***

***}***

***}***

**Usage**:

* Load the Java configuration in your application:

***ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfig.class);***

***JdbcTemplate jdbcTemplate = context.getBean(JdbcTemplate.class);***

**Notes**:

* Java-based configuration is more concise and type-safe, leveraging Java code and annotations.
* It integrates well with modern Spring applications, especially those using Spring Boot.
* Use @ComponentScan if you need to scan for additional components.

**3. Defining DataSource: (Basic – DriverManagerDataSource and Production-Grade – HikariCP,** **Apache DBCP)**

The DataSource is the core component for managing database connections in Spring JDBC. You can use a basic DataSource for development or a production-grade connection pool for better performance and scalability.

**What is a Connection Pool?**

A **connection pool** is a cache of database connections maintained by an application so that they can be reused when the application needs to interact with the database. Instead of opening and closing a new database connection for every request, the connection pool keeps a set of pre-established connections that can be borrowed, used, and returned to the pool. In the context of JDBC (Java Database Connectivity), a connection pool is typically managed by a library like Apache DBCP, HikariCP, or a container like Tomcat.

**How it Works**

* **Initialization**: When the application starts, the connection pool creates a predefined number of database connections (minimum pool size).
* **Borrowing**: When the application needs to query the database, it borrows a connection from the pool.
* **Usage**: The application uses the connection to execute SQL queries.
* **Returning**: After the query is complete, the connection is returned to the pool (not closed) for reuse.
* **Management**: The pool manages connection lifecycle, including creating new connections if needed (up to a maximum pool size), closing idle connections, and handling stale or broken connections.

**Why is a Connection Pool Needed?**

Connection pools are essential for improving the performance, scalability, and resource efficiency of database-driven applications. Here’s why:

1. **Performance Improvement**:
   * Establishing a new database connection is an expensive operation, involving network communication, authentication, and resource allocation on both the application and database server.
   * Reusing existing connections from a pool eliminates this overhead, significantly reducing latency for database operations.
2. **Resource Efficiency**:
   * Without a connection pool, each request might open a new connection, consuming database resources (memory, threads, sockets) and potentially exhausting the database server's capacity.
   * A connection pool limits the number of open connections, ensuring efficient use of resources on both the application and database sides.
3. **Scalability**:
   * In high-concurrency environments (e.g., web applications with many users), creating a new connection for each request can overwhelm the database, leading to failures or slowdowns.
   * Connection pools allow applications to handle multiple requests with a limited number of connections, improving scalability.
4. **Connection Management**:
   * Connection pools handle tasks like maintaining a minimum and maximum number of connections, closing idle connections, and detecting/replacing broken connections.
   * This reduces the complexity of connection management in the application code.
5. **Reliability**:
   * Pools can validate connections before handing them out (e.g., checking if they’re still alive) and recover from transient database issues, improving application robustness.

**Using DriverManagerDataSource (Basic)**

DriverManagerDataSource is a simple implementation that creates a new connection for each request. It is suitable for development or testing but not recommended for production due to the lack of connection pooling.

**Example: DriverManagerDataSource (XML)**

***<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">***

***<property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>***

***<property name="url" value="jdbc:mysql://localhost:3306/mydb"/>***

***<property name="username" value="root"/>***

***<property name="password" value="password"/>***

***</bean>***

**Example: DriverManagerDataSource (Java)**

***@Bean***

***public DataSource dataSource() {***

***DriverManagerDataSource dataSource = new DriverManagerDataSource();***

***dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");***

***dataSource.setUrl("jdbc:mysql://localhost:3306/mydb");***

***dataSource.setUsername("root");***

***dataSource.setPassword("password");***

***return dataSource;***

***}***

**Notes**:

* Simple to configure but creates a new connection for each request, which is inefficient for high-traffic applications.
* Use for small applications, prototyping, or testing with embedded databases like H2.

**Using HikariCP (Production-Grade)**

HikariCP is a high-performance connection pool, widely used in production environments due to its speed and reliability.

**Maven Dependency**:

***<dependency>***

***<groupId>com.zaxxer</groupId>***

***<artifactId>HikariCP</artifactId>***

***<version>5.1.0</version>***

***</dependency>***

**Example: HikariCP (XML)**

***<bean id="dataSource" class="com.zaxxer.hikari.HikariDataSource">***

***<property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>***

***<property name="jdbcUrl" value="jdbc:mysql://localhost:3306/mydb"/>***

***<property name="username" value="root"/>***

***<property name="password" value="password"/>***

***<property name="maximumPoolSize" value="10"/>***

***<property name="minimumIdle" value="5"/>***

***</bean>***

**Example: HikariCP (Java)**

***@Bean***

***public DataSource dataSource() {***

***HikariDataSource dataSource = new HikariDataSource();***

***dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");***

***dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/mydb");***

***dataSource.setUsername("root");***

***dataSource.setPassword("password");***

***dataSource.setMaximumPoolSize(10);***

***dataSource.setMinimumIdle(5);***

***return dataSource;***

***}***

**Notes**:

* HikariCP is lightweight, fast, and optimized for production use.
* Configure properties like maximumPoolSize and minimumIdle to tune performance based on your application’s needs.

**Using Apache DBCP2 (Production-Grade)**

Apache DBCP2 is another popular connection pooling library, offering robust features for managing database connections.

**Maven Dependency**:

***<dependency>***

***<groupId>org.apache.commons</groupId>***

***<artifactId>commons-dbcp2</artifactId>***

***<version>2.12.0</version>***

***</dependency>***

**Example: Apache DBCP2 (XML)**

***<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">***

***<property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>***

***<property name="url" value="jdbc:mysql://localhost:3306/mydb"/>***

***<property name="username" value="root"/>***

***<property name="password" value="password"/>***

***<property name="maxTotal" value="10"/>***

***<property name="maxIdle" value="5"/>***

***</bean>***

**Example: Apache DBCP2 (Java)**

***@Bean***

***public DataSource dataSource() {***

***BasicDataSource dataSource = new BasicDataSource();***

***dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");***

***dataSource.setUrl("jdbc:mysql://localhost:3306/mydb");***

***dataSource.setUsername("root");***

***dataSource.setPassword("password");***

***dataSource.setMaxTotal(10);***

***dataSource.setMaxIdle(5);***

***return dataSource;***

***}***

**Notes**:

* DBCP2 provides similar functionality to HikariCP but may have slightly higher overhead.
* Configure maxTotal (maximum active connections) and maxIdle (maximum idle connections) for optimal performance.

**Summary**

Setting up a Spring JDBC environment involves:

1. **Adding Dependencies**: Include Spring JDBC, a JDBC driver, and optionally a connection pool (HikariCP or Apache DBCP2) in your Maven/Gradle project.
2. **Configuring Spring**:
   * **XML-Based**: Define DataSource and JdbcTemplate beans in an XML file for legacy or externalized configuration.
   * **Java-Based**: Use @Configuration and @Bean for modern, type-safe configuration.
3. **Defining DataSource**:
   * Use DriverManagerDataSource for simple development or testing.
   * Use HikariCP or Apache DBCP2 for production-grade connection pooling to handle high traffic efficiently.

This setup enables you to use Spring JDBC’s JdbcTemplate for streamlined database operations, with the flexibility to choose the configuration approach and DataSource implementation that best suits your application’s needs.

Before proceeding in the Spring-JDBC, let’s see one application with Raw/traditional JDBC API

**Student Database Application:**

I'll create a console-based CRUD application using Java, H2 database, raw JDBC, and MVC design pattern. The application will manage a simple "Student" entity with ID, name, and email fields.

Here's the complete implementation:

1. First, create a new Maven project and add these dependencies to pom.xml:

***<dependencies>***

***<dependency>***

***<groupId>com.h2database</groupId>***

***<artifactId>h2</artifactId>***

***<version>2.3.230</version>***

***</dependency>***

***</dependencies>***

1. Create the database configuration class package com.example.config;

***import java.sql.Connection;***

***import java.sql.DriverManager;***

***import java.sql.SQLException;***

***public class DatabaseConfig {***

***private static final String JDBC\_URL = "jdbc:h2:mem:studentdb;DB\_CLOSE\_DELAY=-1";***

***private static final String USER = "sa";***

***private static final String PASSWORD = "";***

***public static Connection getConnection() throws SQLException {***

***return DriverManager.getConnection(JDBC\_URL, USER, PASSWORD);***

***}***

***public static void initializeDatabase() {***

***try (Connection conn = getConnection();***

***var stmt = conn.createStatement()) {***

***String sql = """***

***CREATE TABLE IF NOT EXISTS student (***

***id INT PRIMARY KEY AUTO\_INCREMENT,***

***name VARCHAR(100) NOT NULL,***

***email VARCHAR(100) NOT NULL UNIQUE***

***)***

***""";***

***stmt.execute(sql);***

***} catch (SQLException e) {***

***System.err.println("Error initializing database: " + e.getMessage());***

***}***

***}***

***}***

1. Create the model class (src/main/java/com/example/model/Student.java):

***package com.example.model;***

***public class Student {***

***private int id;***

***private String name;***

***private String email;***

***public Student() {}***

***public Student(int id, String name, String email) {***

***this.id = id;***

***this.name = name;***

***this.email = email;***

***}***

***// Getters and setters***

***public int getId() { return id; }***

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

***public String getName() { return name; }***

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

***public String getEmail() { return email; }***

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

***@Override***

***public String toString() {***

***return "Student{id=" + id + ", name='" + name + "', email='" + email + "'}";***

***}***

***}***

1. Create the DAO interface (src/main/java/com/example/dao/StudentDAO.java):

***package com.example.dao;***

***import com.example.model.Student;***

***import java.util.List;***

***public interface StudentDAO {***

***void create(Student student);***

***Student read(int id);***

***void update(Student student);***

***void delete(int id);***

***List<Student> getAll();***

***}***

1. Create the DAO implementation (src/main/java/com/example/dao/StudentDAOImpl.java):

***package com.example.dao;***

***import com.example.config.DatabaseConfig;***

***import com.example.model.Student;***

***import java.sql.\*;***

***import java.util.ArrayList;***

***import java.util.List;***

***public class StudentDAOImpl implements StudentDAO {***

***@Override***

***public void create(Student student) {***

***String sql = "INSERT INTO student (name, email) VALUES (?, ?)";***

***try (Connection conn = DatabaseConfig.getConnection();***

***PreparedStatement pstmt = conn.prepareStatement(sql)) {***

***pstmt.setString(1, student.getName());***

***pstmt.setString(2, student.getEmail());***

***pstmt.executeUpdate();***

***} catch (SQLException e) {***

***System.err.println("Error creating student: " + e.getMessage());***

***}***

***}***

***@Override***

***public Student read(int id) {***

***String sql = "SELECT \* FROM student WHERE id = ?";***

***try (Connection conn = DatabaseConfig.getConnection();***

***PreparedStatement pstmt = conn.prepareStatement(sql)) {***

***pstmt.setInt(1, id);***

***ResultSet rs = pstmt.executeQuery();***

***if (rs.next()) {***

***return new Student(***

***rs.getInt("id"),***

***rs.getString("name"),***

***rs.getString("email")***

***);***

***}***

***} catch (SQLException e) {***

***System.err.println("Error reading student: " + e.getMessage());***

***}***

***return null;***

***}***

***@Override***

***public void update(Student student) {***

***String sql = "UPDATE student SET name = ?, email = ? WHERE id = ?";***

***try (Connection conn = DatabaseConfig.getConnection();***

***PreparedStatement pstmt = conn.prepareStatement(sql)) {***

***pstmt.setString(1, student.getName());***

***pstmt.setString(2, student.getEmail());***

***pstmt.setInt(3, student.getId());***

***pstmt.executeUpdate();***

***} catch (SQLException e) {***

***System.err.println("Error updating student: " + e.getMessage());***

***}***

***}***

***@Override***

***public void delete(int id) {***

***String sql = "DELETE FROM student WHERE id = ?";***

***try (Connection conn = DatabaseConfig.getConnection();***

***PreparedStatement pstmt = conn.prepareStatement(sql)) {***

***pstmt.setInt(1, id);***

***pstmt.executeUpdate();***

***} catch (SQLException e) {***

***System.err.println("Error deleting student: " + e.getMessage());***

***}***

***}***

***@Override***

***public List<Student> getAll() {***

***List<Student> students = new ArrayList<>();***

***String sql = "SELECT \* FROM student";***

***try (Connection conn = DatabaseConfig.getConnection();***

***Statement stmt = conn.createStatement();***

***ResultSet rs = stmt.executeQuery(sql)) {***

***while (rs.next()) {***

***students.add(new Student(***

***rs.getInt("id"),***

***rs.getString("name"),***

***rs.getString("email")***

***));***

***}***

***} catch (SQLException e) {***

***System.err.println("Error retrieving students: " + e.getMessage());***

***}***

***return students;***

***}***

***}***

1. Create the controller (src/main/java/com/example/controller/StudentController.java):

***package com.example.controller;***

***import com.example.dao.StudentDAO;***

***import com.example.dao.StudentDAOImpl;***

***import com.example.model.Student;***

***import java.util.List;***

***public class StudentController {***

***private final StudentDAO studentDAO;***

***public StudentController() {***

***this.studentDAO = new StudentDAOImpl();***

***}***

***public void createStudent(String name, String email) {***

***Student student = new Student(0, name, email);***

***studentDAO.create(student);***

***System.out.println("Student created successfully");***

***}***

***public Student getStudent(int id) {***

***Student student = studentDAO.read(id);***

***if (student == null) {***

***System.out.println("Student not found");***

***}***

***return student;***

***}***

***public void updateStudent(int id, String name, String email) {***

***Student student = new Student(id, name, email);***

***studentDAO.update(student);***

***System.out.println("Student updated successfully");***

***}***

***public void deleteStudent(int id) {***

***studentDAO.delete(id);***

***System.out.println("Student deleted successfully");***

***}***

***public List<Student> getAllStudents() {***

***return studentDAO.getAll();***

***}***

***}***

1. Create the view (src/main/java/com/example/view/StudentView.java):

***package com.example.view;***

***import com.example.controller.StudentController;***

***import com.example.model.Student;***

***import java.util.Scanner;***

***public class StudentView {***

***private final StudentController controller;***

***private final Scanner scanner;***

***public StudentView() {***

***this.controller = new StudentController();***

***this.scanner = new Scanner(System.in);***

***}***

***public void displayMenu() {***

***while (true) {***

***System.out.println("\nStudent Management System");***

***System.out.println("1. Create Student");***

***System.out.println("2. View Student");***

***System.out.println("3. Update Student");***

***System.out.println("4. Delete Student");***

***System.out.println("5. List All Students");***

***System.out.println("6. Exit");***

***System.out.print("Choose an option: ");***

***int choice = scanner.nextInt();***

***scanner.nextLine(); // Consume newline***

***switch (choice) {***

***case 1 -> createStudent();***

***case 2 -> viewStudent();***

***case 3 -> updateStudent();***

***case 4 -> deleteStudent();***

***case 5 -> listAllStudents();***

***case 6 -> {***

***System.out.println("Exiting...");***

***return;***

***}***

***default -> System.out.println("Invalid option");***

***}***

***}***

***}***

***private void createStudent() {***

***System.out.print("Enter name: ");***

***String name = scanner.nextLine();***

***System.out.print("Enter email: ");***

***String email = scanner.nextLine();***

***controller.createStudent(name, email);***

***}***

***private void viewStudent() {***

***System.out.print("Enter student ID: ");***

***int id = scanner.nextInt();***

***Student student = controller.getStudent(id);***

***if (student != null) {***

***System.out.println(student);***

***}***

***}***

***private void updateStudent() {***

***System.out.print("Enter student ID: ");***

***int id = scanner.nextInt();***

***scanner.nextLine(); // Consume newline***

***System.out.print("Enter new name: ");***

***String name = scanner.nextLine();***

***System.out.print("Enter new email: ");***

***String email = scanner.nextLine();***

***controller.updateStudent(id, name, email);***

***}***

***private void deleteStudent() {***

***System.out.print("Enter student ID: ");***

***int id = scanner.nextInt();***

***controller.deleteStudent(id);***

***}***

***private void listAllStudents() {***

***System.out.println("\nAll Students:");***

***for (Student student : controller.getAllStudents()) {***

***System.out.println(student);***

***}***

***}***

***}***

1. Create the main application class (src/main/java/com/example/Main.java):

***package com.example;***

***import com.example.config.DatabaseConfig;***

***import com.example.view.StudentView;***

***public class Main {***

***public static void main(String[] args) {***

***DatabaseConfig.initializeDatabase();***

***StudentView view = new StudentView();***

***view.displayMenu();***

***}***

***}***

Output:

Student Management System

1. Create Student

2. View Student

3. Update Student

4. Delete Student

5. List All Students

6. Exit

Choose an option: 1

Enter name: amol

Enter email: amc@gmail.com

Student created successfully

Student Management System

1. Create Student

2. View Student

3. Update Student

4. Delete Student

5. List All Students

6. Exit

Choose an option: 2

Enter student ID: 1

Student{id=1, name='amol', email='amc@gmail.com'}

Student Management System

1. Create Student

2. View Student

3. Update Student

4. Delete Student

5. List All Students

6. Exit

Choose an option: 3

Enter student ID: 1

Enter new name: amol chougule

Enter new email: amolc@yahoo.com

Student updated successfully

Student Management System

1. Create Student

2. View Student

3. Update Student

4. Delete Student

5. List All Students

6. Exit

Choose an option: 2

Enter student ID: 1

Student{id=1, name='amol chougule', email='amolc@yahoo.com'}

Student Management System

1. Create Student

2. View Student

3. Update Student

4. Delete Student

5. List All Students

6. Exit

Choose an option: 1

Enter name: a

Enter email: a

Student created successfully

Student Management System

1. Create Student

2. View Student

3. Update Student

4. Delete Student

5. List All Students

6. Exit

Choose an option: 2

Enter student ID: 2

Student{id=2, name='a', email='a'}

Student Management System

1. Create Student

2. View Student

3. Update Student

4. Delete Student

5. List All Students

6. Exit

Choose an option: 5

All Students:

Student{id=1, name='amol chougule', email='amolc@yahoo.com'}

Student{id=2, name='a', email='a'}

Student Management System

1. Create Student

2. View Student

3. Update Student

4. Delete Student

5. List All Students

6. Exit

Choose an option: 4

Enter student ID: 2

Student deleted successfully

Student Management System

1. Create Student

2. View Student

3. Update Student

4. Delete Student

5. List All Students

6. Exit

Choose an option: 5

All Students:

Student{id=1, name='amol chougule', email='amolc@yahoo.com'}

Student Management System

1. Create Student

2. View Student

3. Update Student

4. Delete Student

5. List All Students

6. Exit

Choose an option: