

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>
```



```
</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.

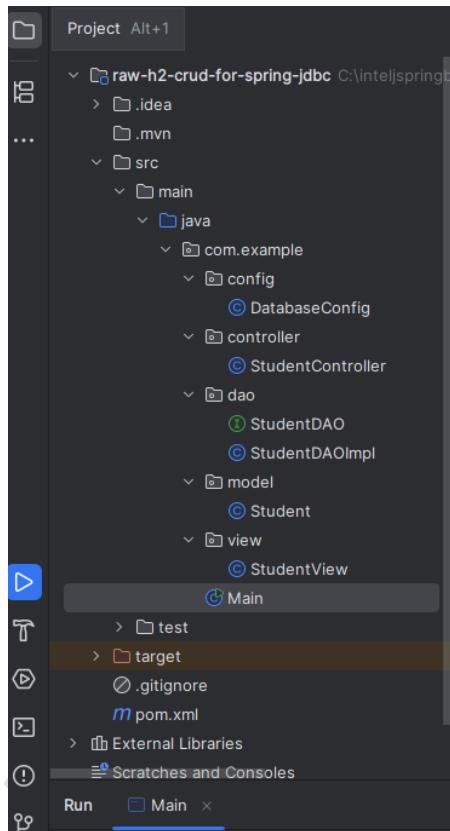
This setup enables you to use Spring JDBC's JdbcTemplate for streamlined database operations, with the flexibility to choose the configuration approach and DataSource implementation that best suits your application's needs.



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>
```

2. 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());
    }
}
}
```

3. 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 + "'}";
    }
}
```



4. 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();
}
```

5. 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.out.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"));
            }
        }
    }
}
```



```
        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 getting all students: " + e.getMessage());
    }
    return students;
}
```



```
        }
    } catch (SQLException e) {
        System.err.println("Error retrieving students: " + e.getMessage());
    }
    return students;
}
}
```

6. 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();
    }
}
```

7. 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);
    }
}
```

8. 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{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: 5

- Student Management System
1. Create Student
 2. View Student
 3. Update Student
 4. Delete Student
 5. List All Students
 6. Exit
- Choose an option:

All Students:

In above application, if you want to use the MySQL Database then, make some changes in above program.

- Create the database in MySQL: CREATE DATABASE studentdb;
- Replace the DatabaseConfig.java with MySqlDatabaseConfig.java
- From StudentDAOImpl.java replace all instances of DatabaseConfig with MySqlDatabaseConfig

MySqlDatabaseConfig.java file:

```
package com.example.config;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class MySqlDatabaseConfig {
    private static final String JDBC_URL = "jdbc:mysql://localhost:3306/studentdb";
    private static final String USER = "root"; // Replace with your MySQL username
    private static final String PASSWORD = "Archer@12345"; // Replace with your MySQL 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);
            System.out.println("Table 'student' created or already exists.");
        } catch (SQLException e) {

```



```

        System.err.println("Error initializing database: " + e.getMessage());
        e.printStackTrace(); // Print stack trace for debugging
        throw new RuntimeException("Failed to initialize database", e);
    }
}
}
}

```

Same Application Can be Written in the Spring-Context (without JdbcTemplate)

Below, I'll outline the changes needed, focusing on:

1. Switching from H2 to MySQL.
2. Adding Spring Core for dependency injection.
3. Using Spring JDBC's DataSource without JdbcTemplate for database operations.
4. Ensuring the MVC structure remains intact.

Prerequisites

- A MySQL server running locally or remotely.
- The studentdb database created in MySQL:

CREATE DATABASE studentdb;

- MySQL credentials (username and password) configured correctly.

Changes to the Application

1. Update Maven Dependencies

Update pom.xml to include MySQL Connector/J and Spring Core dependencies, and remove the H2 dependency. Since we're using Spring JDBC without JdbcTemplate, we only need the Spring Context module for dependency injection and Spring JDBC for DataSource support.

Updated pom.xml:

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>student-crud</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>

  <dependencies>
    <!-- MySQL Connector -->
    <dependency>

```



```
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>8.0.33</version>
</dependency>
<!-- Spring Core and JDBC -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.39</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.3.39</version>
</dependency>
</dependencies>
</project>
```

Changes:

- Removed H2 dependency (com.h2database:h2).
- Added MySQL Connector/J for MySQL support.
- Added spring-context for dependency injection and spring-jdbc for DataSource management.

2. Configure Spring Context

Create a Spring configuration class to set up the DataSource bean and wire dependencies for the MVC components. This replaces the DatabaseConfig class's raw JDBC connection logic.

New src/main/java/com/example/config/SpringConfig.java:

```
package com.example.config;

import com.example.controller.StudentController;
import com.example.dao.StudentDAO;
import com.example.dao.StudentDAOImpl;
import com.example.view.StudentView;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
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/studentdb?useSSL=false&serverTimezone=UTC");
    }
}
```



```
    dataSource.setUsername("root"); // Replace with your MySQL username
    dataSource.setPassword("your_password"); // Replace with your MySQL password
    return dataSource;
}

@Bean
public StudentDAO studentDAO(DataSource dataSource) {
    return new StudentDAOImpl(dataSource);
}

@Bean
public StudentController studentController(StudentDAO studentDAO) {
    return new StudentController(studentDAO);
}

@Bean
public StudentView studentView(StudentController studentController) {
    return new StudentView(studentController);
}

@Bean
public DatabaseInitializer databaseInitializer(DataSource dataSource) {
    return new DatabaseInitializer(dataSource);
}
}
```

Explanation:

- Defines a DataSource bean using Spring's DriverManagerDataSource (a simple implementation for development; consider HikariCP for production).
- Configures MySQL connection details (URL, username, password).
- Creates beans for StudentDAO, StudentController, and StudentView, injecting dependencies appropriately.
- Adds a DatabaseInitializer bean (defined below) to handle table creation.

3. Create Database Initializer

Since we're not using JdbcTemplate, create a separate class to initialize the student table using raw JDBC with Spring's DataSource.

New src/main/java/com/example/config/DatabaseInitializer.java:

```
package com.example.config;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;

public class DatabaseInitializer {
    private final DataSource dataSource;
```



```

public DatabaseInitializer(DataSource dataSource) {
    this.dataSource = dataSource;
    initialize();
}

private void initialize() {
    try (Connection conn = dataSource.getConnection();
        Statement 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);
        System.out.println("Table 'student' created or already exists.");
    } catch (SQLException e) {
        System.err.println("Error initializing database: " + e.getMessage());
        throw new RuntimeException("Failed to initialize database", e);
    }
}
}
}

```

Explanation:

- Takes a DataSource via constructor injection.
- Executes the table creation query using raw JDBC (Connection and Statement).
- Called automatically when the Spring context initializes the DatabaseInitializer bean.

4. Update StudentDAO Interface

The StudentDAO interface remains unchanged, but we'll note it here for completeness:

```

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();
}

```

5. Update StudentDAOImpl to Use Spring DataSource

Modify StudentDAOImpl to use Spring's DataSource instead of raw DriverManager.getConnection(). We'll continue using raw JDBC (Connection and PreparedStatement) instead of JdbcTemplate.



Updated src/main/java/com/example/dao/StudentDAOImpl.java:

```
package com.example.dao;
import com.example.model.Student;
import javax.sql.DataSource;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class StudentDAOImpl implements StudentDAO {
    private final DataSource dataSource;

    public StudentDAOImpl(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Override
    public void create(Student student) {
        String sql = "INSERT INTO student (name, email) VALUES (?, ?)";
        try (Connection conn = dataSource.getConnection()) {
            PreparedStatement pstmt = conn.prepareStatement(sql));
            pstmt.setString(1, student.getName());
            pstmt.setString(2, student.getEmail());
            pstmt.executeUpdate();
            System.out.println("Student created successfully in DAO.");
        } catch (SQLException e) {
            System.err.println("Error creating student: " + e.getMessage());
            throw new RuntimeException("Failed to create student", e);
        }
    }

    @Override
    public Student read(int id) {
        String sql = "SELECT * FROM student WHERE id = ?";
        try (Connection conn = dataSource.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());
            throw new RuntimeException("Failed to read student", e);
        }
    }
}
```



```
        return null;
    }

@Override
public void update(Student student) {
    String sql = "UPDATE student SET name = ?, email = ? WHERE id = ?";
    try (Connection conn = dataSource.getConnection()) {
        PreparedStatement pstmt = conn.prepareStatement(sql)) {
            pstmt.setString(1, student.getName());
            pstmt.setString(2, student.getEmail());
            pstmt.setInt(3, student.getId());
            pstmt.executeUpdate();
            System.out.println("Student updated successfully in DAO.");
    } catch (SQLException e) {
        System.err.println("Error updating student: " + e.getMessage());
        throw new RuntimeException("Failed to update student", e);
    }
}

@Override
public void delete(int id) {
    String sql = "DELETE FROM student WHERE id = ?";
    try (Connection conn = dataSource.getConnection()) {
        PreparedStatement pstmt = conn.prepareStatement(sql)) {
            pstmt.setInt(1, id);
            pstmt.executeUpdate();
            System.out.println("Student deleted successfully in DAO.");
    } catch (SQLException e) {
        System.err.println("Error deleting student: " + e.getMessage());
        throw new RuntimeException("Failed to delete student", e);
    }
}

@Override
public List<Student> getAll() {
    List<Student> students = new ArrayList<>();
    String sql = "SELECT * FROM student";
    try (Connection conn = dataSource.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());
        throw new RuntimeException("Failed to retrieve students", e);
    }
    return students;
}
}
```

Changes:

- Added a constructor to accept a DataSource injected by Spring.
- Replaced DatabaseConfig.getConnection() with dataSource.getConnection().
- Kept raw JDBC operations (Connection, PreparedStatement, Statement) instead of using JdbcTemplate.
- Improved error handling by throwing RuntimeException for SQL errors to make issues more visible.

6. Update StudentController

Modify StudentController to accept StudentDAO via constructor injection, aligning with Spring's dependency injection.

Updated src/main/java/com/example/controller/StudentController.java:

```
package com.example.controller;
import com.example.dao.StudentDAO;
import com.example.model.Student;
import java.util.List;

public class StudentController {
    private final StudentDAO studentDAO;

    public StudentController(StudentDAO studentDAO) {
        this.studentDAO = studentDAO;
    }

    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);
    }
}
```



```

        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();
    }
}

```

Changes:

- Replaced manual instantiation of StudentDAOImpl with constructor injection.
- No logic changes, as the controller delegates to the DAO.

7. Update StudentView

Modify StudentView to accept StudentController via constructor injection.

Updated 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(StudentController controller) {
        this.controller = controller;
        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);
    }
}
}

```

Changes:

- Replaced manual instantiation of StudentController with constructor injection.
- No changes to the console interface or logic.

8. Update Main Class

Modify the Main class to use Spring's ApplicationContext to initialize the application and retrieve the StudentView bean.

Updated src/main/java/com/example/Main.java:

```

package com.example;

import com.example.config.SpringConfig;
import com.example.view.StudentView;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
    public static void main(String[] args) {
        // Initialize Spring context
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(SpringConfig.class);

        // Get StudentView bean and start the application
        StudentView view = context.getBean(StudentView.class);
        view.displayMenu();

        // Close context when done
        context.close();
    }
}

```

Changes:

- Replaced manual instantiation with Spring's AnnotationConfigApplicationContext.
- The DatabaseInitializer bean is automatically created during context initialization, so we don't need to call initializeDatabase() explicitly.

9. Keep Student Model Unchanged

The Student model class remains unchanged, as it's a simple POJO and independent of the database or Spring.



src/main/java/com/example/model/Student.java (unchanged):

```
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 + "'}";
    }
}
```

10. Delete Obsolete DatabaseConfig

Remove the original DatabaseConfig.java, as it's replaced by SpringConfig and DatabaseInitializer.

Above Application Using JdbcTemplate in Spring Core:

The console-based CRUD application (previously modified to use MySQL with Spring Core and Spring JDBC without JdbcTemplate) to use **Spring's JdbcTemplate** for database operations. The application will continue to use MySQL as the database, maintain the MVC pattern, and leverage Spring Core for dependency injection. Using JdbcTemplate simplifies database operations by reducing boilerplate code for handling Connection, PreparedStatement, and ResultSet objects, while providing robust error handling and resource management.

Below, I'll provide the complete rewritten application, focusing on changes to integrate JdbcTemplate. The structure remains similar to the previous version, with updates to StudentDAOImpl to use JdbcTemplate, and minor adjustments to other components to align with Spring best practices.

Project Overview

- **Database:** MySQL (studentdb database, student table with id, name, email).
- **Framework:** Spring Core with Spring JDBC, using JdbcTemplate.



- **Pattern:** MVC (Model: Student, View: StudentView, Controller: StudentController).
- **DAO:** StudentDAO and StudentDAOImpl for data access, now using JdbcTemplate.

Prerequisites

- MySQL server running with the studentdb database created:

```
CREATE DATABASE studentdb;
```

- MySQL credentials configured correctly.
- Maven setup for dependency management.

Complete Application Code

1. Maven Dependencies (`pom.xml`)

The dependencies remain the same as the previous Spring-based version, as JdbcTemplate is included in the spring-jdbc module.

`pom.xml:`

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>student-crud</artifactId>
  <version>1.0-SNAPSHOT</version>
  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>

  <dependencies>
    <!-- MySQL Connector -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.33</version>
    </dependency>
    <!-- Spring Core and JDBC -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.3.39</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>5.3.39</version>
    </dependency>
  </dependencies>
</project>
```



2. Spring Configuration (SpringConfig.java)

The configuration class sets up the DataSource, JdbcTemplate, and MVC component beans. We add a JdbcTemplate bean to be injected into the DAO.

src/main/java/com/example/config/SpringConfig.java:

```
package com.example.config;

import com.example.controller.StudentController;
import com.example.dao.StudentDAO;
import com.example.dao.StudentDAOImpl;
import com.example.view.StudentView;
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/studentdb?useSSL=false&serverTimezone=UTC");
        dataSource.setUsername("root"); // Replace with your MySQL username
        dataSource.setPassword("your_password"); // Replace with your MySQL password
        return dataSource;
    }

    @Bean
    public JdbcTemplate jdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }

    @Bean
    public StudentDAO studentDAO(JdbcTemplate jdbcTemplate) {
        return new StudentDAOImpl(jdbcTemplate);
    }

    @Bean
    public StudentController studentController(StudentDAO studentDAO) {
        return new StudentController(studentDAO);
    }
}
```



```
@Bean  
public StudentView studentView(StudentController studentController) {  
    return new StudentView(studentController);  
}  
  
@Bean  
public DatabaseInitializer databaseInitializer(JdbcTemplate jdbcTemplate) {  
    return new DatabaseInitializer(jdbcTemplate);  
}  
}
```

Changes:

- Added a JdbcTemplate bean, initialized with the DataSource.
- Updated studentDAO bean to accept JdbcTemplate instead of DataSource.
- Updated DatabaseInitializer bean to accept JdbcTemplate for table creation.

3. Database Initializer (DatabaseInitializer.java)

Use JdbcTemplate to create the student table during application startup.

src/main/java/com/example/config/DatabaseInitializer.java:

```
package com.example.config;  
import org.springframework.jdbc.core.JdbcTemplate;  
  
public class DatabaseInitializer {  
    private final JdbcTemplate jdbcTemplate;  
    public DatabaseInitializer(JdbcTemplate jdbcTemplate) {  
        this.jdbcTemplate = jdbcTemplate;  
        initialize();  
    }  
  
    private void initialize() {  
        try {  
            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  
            );  
            """;  
            jdbcTemplate.execute(sql);  
            System.out.println("Table 'student' created or already exists.");  
        } catch (Exception e) {  
            System.err.println("Error initializing database: " + e.getMessage());  
            throw new RuntimeException("Failed to initialize database", e);  
        }  
    }  
}
```



Changes:

- Replaced raw JDBC (Connection and Statement) with JdbcTemplate.execute().
- Simplified error handling, as JdbcTemplate manages resources automatically.

4. Student Model (Student.java)

The model remains unchanged, as it's a simple POJO.

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 + "'}";
    }
}
```

5. Student DAO Interface (StudentDAO.java)

The DAO interface remains unchanged.

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();  
}
```

6. Student DAO Implementation (StudentDAOImpl.java)

Rewrite StudentDAOImpl to use JdbcTemplate for all CRUD operations, significantly reducing boilerplate code.

src/main/java/com/example/dao/StudentDAOImpl.java:

```
package com.example.dao;  
  
import com.example.model.Student;  
import org.springframework.jdbc.core.JdbcTemplate;  
import org.springframework.jdbc.core.RowMapper;  
  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import java.util.List;  
  
public class StudentDAOImpl implements StudentDAO {  
    private final JdbcTemplate jdbcTemplate;  
  
    public StudentDAOImpl(JdbcTemplate jdbcTemplate) {  
        this.jdbcTemplate = jdbcTemplate;  
    }  
  
    private final RowMapper<Student> studentRowMapper = (ResultSet rs, int rowNum) -> new  
    Student(  
        rs.getInt("id"),  
        rs.getString("name"),  
        rs.getString("email")  
    );  
  
    @Override  
    public void create(Student student) {  
        String sql = "INSERT INTO student (name, email) VALUES (?, ?)";  
        try {  
            jdbcTemplate.update(sql, student.getName(), student.getEmail());  
            System.out.println("Student created successfully in DAO.");  
        } catch (Exception e) {  
            System.err.println("Error creating student: " + e.getMessage());  
            throw new RuntimeException("Failed to create student", e);  
        }  
    }  
}
```



```
}

@Override
public Student read(int id) {
    String sql = "SELECT * FROM student WHERE id = ?";
    try {
        return jdbcTemplate.queryForObject(sql, new Object[]{id}, studentRowMapper);
    } catch (Exception e) {
        System.err.println("Error reading student: " + e.getMessage());
        return null; // Return null if student not found or error occurs
    }
}

@Override
public void update(Student student) {
    String sql = "UPDATE student SET name = ?, email = ? WHERE id = ?";
    try {
        jdbcTemplate.update(sql, student.getName(), student.getEmail(), student.getId());
        System.out.println("Student updated successfully in DAO.");
    } catch (Exception e) {
        System.err.println("Error updating student: " + e.getMessage());
        throw new RuntimeException("Failed to update student", e);
    }
}

@Override
public void delete(int id) {
    String sql = "DELETE FROM student WHERE id = ?";
    try {
        jdbcTemplate.update(sql, id);
        System.out.println("Student deleted successfully in DAO.");
    } catch (Exception e) {
        System.err.println("Error deleting student: " + e.getMessage());
        throw new RuntimeException("Failed to delete student", e);
    }
}

@Override
public List<Student> getAll() {
    String sql = "SELECT * FROM student";
    try {
        return jdbcTemplate.query(sql, studentRowMapper);
    } catch (Exception e) {
        System.err.println("Error retrieving students: " + e.getMessage());
        throw new RuntimeException("Failed to retrieve students", e);
    }
}
}
```



Changes:

- Injected JdbcTemplate via constructor.
- Defined a RowMapper<Student> to map ResultSet rows to Student objects.
- Replaced raw JDBC operations with JdbcTemplate methods:
 - create: Uses jdbcTemplate.update() for INSERT.
 - read: Uses jdbcTemplate.queryForObject() for SELECT by ID.
 - update: Uses jdbcTemplate.update() for UPDATE.
 - delete: Uses jdbcTemplate.update() for DELETE.
 - getAll: Uses jdbcTemplate.query() for SELECT all.
- Simplified resource management, as JdbcTemplate handles Connection, PreparedStatement, and ResultSet cleanup.
- Kept error handling to log issues and throw RuntimeException for critical errors.

7. Student Controller (StudentController.java)

The controller remains unchanged, as it delegates to the DAO.

`src/main/java/com/example/controller/StudentController.java:`

```
package com.example.controller;

import com.example.dao.StudentDAO;
import com.example.model.Student;
import java.util.List;

public class StudentController {
    private final StudentDAO studentDAO;

    public StudentController(StudentDAO studentDAO) {
        this.studentDAO = studentDAO;
    }

    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();
    }
}
```

8. Student View (StudentView.java)

The view remains unchanged, as it handles console interaction.

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(StudentController controller) {
        this.controller = controller;
        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);  
    }  
}
```

9. Main Class (Main.java)

The main class remains unchanged, using Spring's ApplicationContext to bootstrap the application.

src/main/java/com/example/Main.java:

```
package com.example;  
  
import com.example.config.SpringConfig;  
import com.example.view.StudentView;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
public class Main {  
    public static void main(String[] args) {  
        // Initialize Spring context  
        AnnotationConfigApplicationContext context = new  
        AnnotationConfigApplicationContext(SpringConfig.class);  
  
        // Get StudentView bean and start the application  
        StudentView view = context.getBean(StudentView.class);  
        view.displayMenu();  
  
        // Close context when done  
        context.close();  
    }  
}
```

Key Changes from Previous Version

- **JdbcTemplate Integration:** Replaced raw JDBC (Connection, PreparedStatement, ResultSet) with JdbcTemplate methods in StudentDAOImpl and DatabaseInitializer.
- **Simplified DAO:** Reduced boilerplate code for database operations, as JdbcTemplate handles resource management and parameter binding.
- **RowMapper:** Introduced studentRowMapper to map query results to Student objects.
- **Error Handling:** Kept consistent error logging and exception throwing, leveraging JdbcTemplate's exception translation (e.g., DataAccessException).
- **Configuration:** Added JdbcTemplate bean in SpringConfig and updated dependencies for StudentDAOImpl and DatabaseInitializer.

Benefits of Using JdbcTemplate

- **Less Boilerplate:** Eliminates manual handling of JDBC resources, reducing code verbosity.



- **Automatic Resource Management:** Closes Connection, PreparedStatement, and ResultSet automatically.
 - **Parameter Binding:** Simplifies setting query parameters, preventing SQL injection.
 - **Exception Translation:** Converts SQL exceptions to Spring's DataAccessException hierarchy, improving error handling.
-

Below, I'll rewrite the application to:

1. Use **Spring's annotation-based configuration** with component scanning.
2. Replace manual bean definitions in SpringConfig.java with @Component, @Service, and @Repository annotations where appropriate.
3. Retain JdbcTemplate for database operations.
4. Use an anonymous inner class for RowMapper (per the previous request).
5. Maintain the MVC pattern and MySQL database.

Prerequisites

- MySQL server running with the studentdb database created:

CREATE DATABASE studentdb;

- MySQL credentials configured correctly.
- Maven setup with the same dependencies as before.

Updated Application Code

1. Maven Dependencies (pom.xml)

The dependencies remain unchanged, as they already include MySQL Connector/J and Spring Core/JDBC for JdbcTemplate.

pom.xml:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>student-crud</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>

  <dependencies>
    <!-- MySQL Connector -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.33</version>
    </dependency>
  
```



```
<!-- Spring Core and JDBC -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.39</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.3.39</version>
</dependency>
</dependencies>
</project>
```

2. Spring Configuration (SpringConfig.java)

Modify the configuration to enable component scanning and define only the DataSource and JdbcTemplate beans, as other components will be annotated with @Component, @Service, or @Repository.

```
src/main/java/com/example/config/SpringConfig.java:
package com.example.config;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import javax.sql.DataSource;
@Configuration
@ComponentScan(basePackages = "com.example")
public class SpringConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");

        dataSource.setUrl("jdbc:mysql://localhost:3306/studentdb?useSSL=false&serverTimezone=UTC");
        dataSource.setUsername("root"); // Replace with your MySQL username
        dataSource.setPassword("your_password"); // Replace with your MySQL password
        return dataSource;
    }
    @Bean
    public JdbcTemplate jdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }
}
```

Changes:



- Added @ComponentScan(basePackages = "com.example") to scan for annotated components in the com.example package and its subpackages.
- Removed explicit @Bean definitions for StudentDAO, StudentController, StudentView, and DatabaseInitializer, as these will now be detected via annotations.
- Kept DataSource and JdbcTemplate beans, as they require specific configuration not easily handled by component scanning.

3. Database Initializer (DatabaseInitializer.java)

Annotate DatabaseInitializer with @Component to make it a Spring-managed bean and use @PostConstruct to run initialization after bean creation.

src/main/java/com/example/config/DatabaseInitializer.java:

```
package com.example.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;

@Component
public class DatabaseInitializer {
    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public DatabaseInitializer(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @PostConstruct
    public void initialize() {
        try {
            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
                )
                """;
            jdbcTemplate.execute(sql);
            System.out.println("Table 'student' created or already exists.");
        } catch (Exception e) {
            System.err.println("Error initializing database: " + e.getMessage());
            throw new RuntimeException("Failed to initialize database", e);
        }
    }
}
```



Changes:

- Added `@Component` to make `DatabaseInitializer` a Spring-managed bean.
- Replaced constructor injection with `@Autowired` (equivalent in this context).
- Used `@PostConstruct` to run `initialize()` after the bean is created, ensuring the table is created during application startup.

4. Student Model (Student.java)

The model remains unchanged, as it's a POJO without Spring dependencies.

`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 + "'}";
    }
}
```

5. Student DAO Interface (StudentDAO.java)

The DAO interface remains unchanged.

`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();  
}
```

6. Student DAO Implementation (StudentDAOImpl.java)

Annotate StudentDAOImpl with @Repository and use @Autowired for JdbcTemplate injection. Use an anonymous inner class for RowMapper (per the previous request).

```
src/main/java/com/example/dao/StudentDAOImpl.java:  
package com.example.dao;  
import com.example.model.Student;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.jdbc.core.JdbcTemplate;  
import org.springframework.jdbc.core.RowMapper;  
import org.springframework.stereotype.Repository;  
  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import java.util.List;  
  
@Repository  
public class StudentDAOImpl implements StudentDAO {  
    private final JdbcTemplate jdbcTemplate;  
  
    @Autowired  
    public StudentDAOImpl(JdbcTemplate jdbcTemplate) {  
        this.jdbcTemplate = jdbcTemplate;  
    }  
  
    private final RowMapper<Student> studentRowMapper = new RowMapper<Student>() {  
        @Override  
        public Student mapRow(ResultSet rs, int rowNum) throws SQLException {  
            return new Student(  
                rs.getInt("id"),  
                rs.getString("name"),  
                rs.getString("email")  
            );  
        }  
    };  
  
    @Override  
    public void create(Student student) {  
        String sql = "INSERT INTO student (name, email) VALUES (?, ?)";  
        try {
```



```
        jdbcTemplate.update(sql, student.getName(), student.getEmail());
        System.out.println("Student created successfully in DAO.");
    } catch (Exception e) {
        System.err.println("Error creating student: " + e.getMessage());
        throw new RuntimeException("Failed to create student", e);
    }
}

@Override
public Student read(int id) {
    String sql = "SELECT * FROM student WHERE id = ?";
    try {
        return jdbcTemplate.queryForObject(sql, new Object[]{id}, studentRowMapper);
    } catch (Exception 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 {
        jdbcTemplate.update(sql, student.getName(), student.getEmail(), student.getId());
        System.out.println("Student updated successfully in DAO.");
    } catch (Exception e) {
        System.err.println("Error updating student: " + e.getMessage());
        throw new RuntimeException("Failed to update student", e);
    }
}

@Override
public void delete(int id) {
    String sql = "DELETE FROM student WHERE id = ?";
    try {
        jdbcTemplate.update(sql, id);
        System.out.println("Student deleted successfully in DAO.");
    } catch (Exception e) {
        System.err.println("Error deleting student: " + e.getMessage());
        throw new RuntimeException("Failed to delete student", e);
    }
}

@Override
public List<Student> getAll() {
    String sql = "SELECT * FROM student";
    try {
        return jdbcTemplate.query(sql, studentRowMapper);
```

```
        } catch (Exception e) {
            System.err.println("Error retrieving students: " + e.getMessage());
            throw new RuntimeException("Failed to retrieve students", e);
        }
    }
}
```

Changes:

- Added @Repository to mark StudentDAOImpl as a Spring-managed DAO bean, enabling component scanning and exception translation.
- Used @Autowired for JdbcTemplate injection.
- Kept the RowMapper as an anonymous inner class, as requested.
- No changes to the JdbcTemplate logic, as it was already optimized.

7. Student Controller (StudentController.java)

Annotate StudentController with @Service and use @Autowired for StudentDAO injection.

src/main/java/com/example/controller/StudentController.java:

```
package com.example.controller;
import com.example.dao.StudentDAO;
import com.example.model.Student;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class StudentController {
    private final StudentDAO studentDAO;

    @Autowired
    public StudentController(StudentDAO studentDAO) {
        this.studentDAO = studentDAO;
    }

    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();
}
}
```

Changes:

- Added @Service to mark StudentController as a Spring-managed service bean.
- Used @Autowired for StudentDAO injection.
- No logic changes, as the controller delegates to the DAO.

8. Student View (StudentView.java)

Annotate StudentView with @Component and use @Autowired for StudentController injection.

src/main/java/com/example/view/StudentView.java:

```
package com.example.view;
import com.example.controller.StudentController;
import com.example.model.Student;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.util.Scanner;

@Component
public class StudentView {
    private final StudentController controller;
    private final Scanner scanner;

    @Autowired
    public StudentView(StudentController controller) {
        this.controller = controller;
        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: ");



https://archerinfotech.in
```



```

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);
    }
}
}

```

Changes:

- Added @Component to mark StudentView as a Spring-managed bean.
- Used @Autowired for StudentController injection.
- No changes to the console interface or logic.

9. Main Class (Main.java)

The main class remains unchanged, as it uses Spring's ApplicationContext to bootstrap the application.

```

src/main/java/com/example/Main.java:
package com.example;
import com.example.config.SpringConfig;
import com.example.view.StudentView;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class Main {
    public static void main(String[] args) {
        // Initialize Spring context
        AnnotationConfigApplicationContext context = new
        AnnotationConfigApplicationContext(SpringConfig.class);

        // Get StudentView bean and start the application
        StudentView view = context.getBean(StudentView.class);
        view.displayMenu();

        // Close context when done
        context.close();
    }
}

```

Key Changes from Previous Version

- **Annotation-Based Configuration:**
 - Replaced explicit @Bean definitions for StudentDAO, StudentController, StudentView, and DatabaseInitializer with @Component, @Repository, @Service, and @Component annotations, respectively.
 - Added @ComponentScan to SpringConfig to automatically detect these annotated classes.
 - Used @Autowired for dependency injection instead of manual bean wiring.
 - Added @PostConstruct in DatabaseInitializer to ensure table creation after bean initialization.
- **RowMapper:** Retained the anonymous inner class for RowMapper<Student>, as requested, avoiding the lambda function.
- **JdbcTemplate:** Kept the JdbcTemplate-based DAO operations from the previous version.
- **MVC Structure:** Preserved, with dependencies now managed via annotations.
- **MySQL:** Continued using MySQL with the same table structure and configuration.

Benefits of Annotation-Based Configuration

- **Reduced Boilerplate:** Eliminated manual @Bean definitions for most components, making the configuration more concise.
 - **Automatic Discovery:** Component scanning detects @Component, @Service, and @Repository beans, simplifying maintenance.
 - **Spring Best Practices:** Using @Repository enables Spring's exception translation for database operations, and @Service clearly marks business logic components.
 - **Flexibility:** Annotations make it easier to add new components without modifying SpringConfig.
-

Logging with SLF4J and Logback:

SLF4J (Simple Logging Facade for Java)

- **What it is:** SLF4J is a logging facade, meaning it provides a common API for logging that abstracts the underlying logging framework. It allows developers to write log statements without being tied to a specific logging implementation (e.g., Logback, Log4j, or Java Util Logging).
- **Purpose:** Enables flexibility to switch logging frameworks without changing application code. For example, you can use SLF4J with Logback today and switch to Log4j later by changing configuration and dependencies.
- **Key Features:**
 - Simple, lightweight API.
 - Supports parameterized logging for better performance (e.g., logger.info("User {} logged in", userId)).
 - Acts as a bridge between different logging frameworks.
- **How it works:** You include SLF4J in your project and a binding for the desired logging framework (e.g., SLF4J-Logback binding). SLF4J delegates log calls to the bound framework.
- **Dependency Example (Maven, for SLF4J API):**

```
<dependency>
  <groupId>org.slf4j</groupId>
```



```
<artifactId>slf4j-api</artifactId>
<version>2.0.16</version> <!-- Check for latest version -->
</dependency>
```

Logback

- **What it is:** Logback is a logging framework, often used as the backend implementation for SLF4J. It's the successor to Log4j 1.x, designed to be faster, more reliable, and feature-rich.
- **Purpose:** Handles the actual logging (writing logs to console, files, databases, etc.) based on configuration. It's commonly paired with SLF4J for a complete logging solution.
- **Key Features:**
 - Configurable via XML or Groovy (logback.xml).
 - Supports multiple appenders (e.g., console, file, rolling file, database).
 - Automatic log file rotation and compression.
 - Filtering based on log levels (e.g., DEBUG, INFO, ERROR).
 - High performance and low memory footprint.
- **How it works:** Logback processes log messages from SLF4J, formats them, and directs them to configured outputs. For example, you can configure it to write DEBUG logs to a file and ERROR logs to both console and email.
- **Dependency Example (Maven, for Logback with SLF4J):**

```
<dependency>
<groupId>ch.qos.logback</groupId>
<artifactId>logback-classic</artifactId>
<version>1.5.12</version> <!-- Check for latest version -->
</dependency>
```

- The logback-classic module includes SLF4J binding and core functionality. You also need logback-core (usually included transitively).

SLF4J + Logback Together

- SLF4J provides the API for logging in your code, while Logback is the implementation that processes and outputs logs.
- Example code:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MyClass {
    private static final Logger logger = LoggerFactory.getLogger(MyClass.class);

    public void doSomething() {
        logger.info("Processing data for user: {}", userId);
        try {
            // Some operation
        } catch (Exception e) {
            logger.error("Error occurred", e);
        }
    }
}
```



- Configuration example (logback.xml):

```

<configuration>
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>
    <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>logs/app.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>logs/app.%d{yyyy-MM-dd}.log</fileNamePattern>
            <maxHistory>30</maxHistory>
        </rollingPolicy>
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>
    <root level="INFO">
        <appender-ref ref="CONSOLE" />
        <appender-ref ref="FILE" />
    </root>
</configuration>

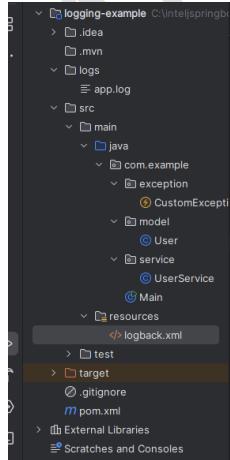
```

Why Use SLF4J with Logback?

- **SLF4J**: Provides a consistent API, making your code portable across logging frameworks.
- **Logback**: Offers robust, performant logging with advanced features like rolling files and asynchronous logging.
- Together, they're a popular choice for Java applications due to ease of use, flexibility, and performance.

Example: a complete example of a Java project with a proper logging setup using SLF4J with Logback, including a well-organized project structure, complete code, and configuration files.

Project Structure



1. pom.xml (Maven Build File)

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>my-logging-project</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <slf4j.version>2.0.16</slf4j.version>
    <logback.version>1.5.12</logback.version>
    <junit.version>5.11.3</junit.version>
  </properties>

  <dependencies>
    <!-- SLF4J API -->
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>${slf4j.version}</version>
    </dependency>
    <!-- Logback Classic (SLF4J Implementation) -->
    <dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
      <version>${logback.version}</version>
    </dependency>
    <!-- JUnit for testing -->
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>${junit.version}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
```



```
<version>3.13.0</version>
<configuration>
    <source>${maven.compiler.source}</source>
    <target>${maven.compiler.target}</target>
</configuration>
</plugin>
</plugins>
</build>
</project>
```

2. src/main/resources/logback.xml (Logging Configuration)

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <!-- Console Appender -->
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n</pattern>
        </encoder>
    </appender>

    <!-- File Appender with Rolling Policy -->
    <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>logs/app.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>logs/app.%d{yyyy-MM-dd}.log</fileNamePattern>
            <maxHistory>30</maxHistory>
            <totalSizeCap>3GB</totalSizeCap>
        </rollingPolicy>
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n</pattern>
        </encoder>
    </appender>

    <!-- Async Appender for better performance -->
    <appender name="ASYNC" class="ch.qos.logback.classic.AsyncAppender">
        <appender-ref ref="FILE" />
        <queueSize>500</queueSize>
        <discardingThreshold>0</discardingThreshold>
    </appender>

    <!-- Root Logger Configuration -->
    <root level="INFO">
        <appender-ref ref="CONSOLE" />
        <appender-ref ref="ASYNC" />
    </root>
```



```
<!-- Specific Logger for Debugging -->
<logger name="com.example.service" level="DEBUG" additivity="false">
    <appender-ref ref="CONSOLE" />
    <appender-ref ref="ASYNC" />
</logger>
</configuration>
```

3. src/main/java/com/example/model/User.java

```
package com.example.model;

public class User {
    private final String id;
    private final String name;

    public User(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public String getId() {
        return id;
    }

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

4. src/main/java/com/example/exception/CustomException.java

```
package com.example.exception;

public class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }

    public CustomException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

5. src/main/java/com/example/service/UserService.java

```
package com.example.service;

import com.example.exception.CustomException;
```



```
import com.example.model.User;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class UserService {
    private static final Logger logger = LoggerFactory.getLogger(UserService.class);

    public User processUser(String userId, String name) throws CustomException {
        logger.trace("Entering processUser with userId: {}, name: {}", userId, name);

        if (userId == null || userId.trim().isEmpty()) {
            logger.warn("Invalid userId provided: {}", userId);
            throw new CustomException("User ID cannot be null or empty");
        }

        try {
            logger.debug("Creating user object for userId: {}", userId);
            User user = new User(userId, name);
            logger.info("Successfully processed user: {}", user.getId());
            return user;
        } catch (Exception e) {
            logger.error("Failed to process user with userId: {}", userId, e);
            throw new CustomException("Error processing user", e);
        } finally {
            logger.trace("Exiting processUser method");
        }
    }
}
```

6. src/main/java/com/example/Main.java

```
package com.example;

import com.example.exception.CustomException;
import com.example.model.User;
import com.example.service.UserService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Main {
    private static final Logger logger = LoggerFactory.getLogger(Main.class);

    public static void main(String[] args) {
        logger.info("Application started");

        UserService userService = new UserService();

        try {
            // Successful case

```



```
User user = userService.processUser("123", "John Doe");
logger.info("Created user: {} with name: {}", user.getId(), user.getName());

// Error case
userService.processUser(null, "Jane Doe");
} catch (CustomException e) {
    logger.error("Application error occurred", e);
}

logger.info("Application shutdown");
}
```

Logging Features:

- Uses SLF4J with Logback implementation
- Multiple logging levels (TRACE, DEBUG, INFO, WARN, ERROR)
- Console and file appenders
- Async appender for better performance
- Daily rolling file policy with 30-day retention
- Detailed log pattern with timestamp, thread, level, and logger name
 - **TRACE:** Very detailed information for fine-grained debugging, typically used during development.
 - **DEBUG:** Detailed information for debugging, less verbose than TRACE, useful for troubleshooting.
 - **INFO:** General information about application progress, indicating normal operation.
 - **WARN:** Indicates potential issues or situations that might cause problems but aren't errors.
 - **ERROR:** Serious issues that indicate a failure in the application, requiring attention.

Logback Configuration:

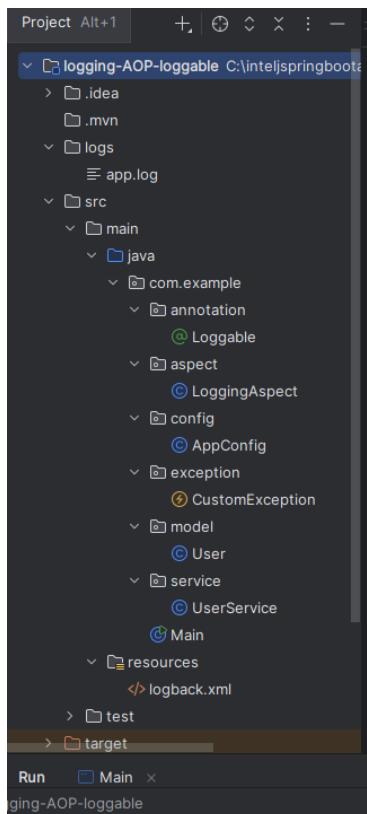
- Console output for immediate feedback
 - File output with rolling policy
 - Async logging to prevent blocking
 - Specific logger for service package with DEBUG level
 - Root logger set to INFO level
-

Logging using @Loggable:

To demonstrate the use of the `@Loggable` annotation, I'll modify the previous program to include a custom `@Loggable` annotation that automatically logs method entry, exit, and exceptions using Aspect-Oriented Programming (AOP) with Spring and SLF4J. This approach reduces boilerplate logging code in the business logic. Since the original program uses SLF4J with Logback, I'll integrate the annotation while maintaining the same logging setup and project structure, adding Spring AOP for the annotation processing.

Overview of Changes

- Introduce a custom @Loggable annotation.
- Use Spring Boot with AOP to process the annotation.
- Modify the project to include Spring dependencies.
- Update the UserService to use @Loggable instead of manual logging.
- Keep the same project structure, logging configuration, and functionality.
- Adjust the Main class to use a Spring application context.



1. pom.xml (Updated for spring-core)

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>my-logging-project</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <spring.version>6.1.14</spring.version>
    <slf4j.version>2.0.16</slf4j.version>
  
```



```
<logback.version>1.5.12</logback.version>
<junit.version>5.11.3</junit.version>
</properties>

<dependencies>
    <!-- Spring Core -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <!-- Spring Context -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <!-- Spring AOP -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aop</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <!-- AspectJ Weaver for AOP -->
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.9.22.1</version>
    </dependency>
    <!-- SLF4J API -->
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>${slf4j.version}</version>
    </dependency>
    <!-- Logback Classic -->
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>${logback.version}</version>
    </dependency>
    <!-- JUnit for Testing -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>${junit.version}</version>
        <scope>test</scope>
    </dependency>
```



```
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.13.0</version>
      <configuration>
        <source>${maven.compiler.source}</source>
        <target>${maven.compiler.target}</target>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

2. src/main/resources/logback.xml (Unchanged)

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
      %msg%n</pattern>
    </encoder>
  </appender>

  <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>logs/app.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <fileNamePattern>logs/app.%d{yyyy-MM-dd}.log</fileNamePattern>
      <maxHistory>30</maxHistory>
      <totalSizeCap>3GB</totalSizeCap>
    </rollingPolicy>
    <encoder>
      <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
      %msg%n</pattern>
    </encoder>
  </appender>

  <appender name="ASYNC" class="ch.qos.logback.classic.AsyncAppender">
    <appender-ref ref="FILE" />
    <queueSize>500</queueSize>
    <discardingThreshold>0</discardingThreshold>
  </appender>

  <root level="INFO">
    <appender-ref ref="CONSOLE" />
```



```
<appender-ref ref="ASYNC" />
</root>

<logger name="com.example.service" level="DEBUG" additivity="false">
    <appender-ref ref="CONSOLE" />
    <appender-ref ref="ASYNC" />
</logger>
</configuration>
```

3. src/main/java/com/example/config/AppConfig.java (New)

```
package com.example.config;

import com.example.aspect.LoggingAspect;
import com.example.service.UserService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@ComponentScan(basePackages = "com.example")
@EnableAspectJAutoProxy
public class AppConfig {
    @Bean
    public UserService userService() {
        return new UserService();
    }

    @Bean
    public LoggingAspect loggingAspect() {
        return new LoggingAspect();
    }
}
```

4. src/main/java/com/example/annotation/Loggable.java (Unchanged)

```
package com.example.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Loggable {
}
```



5. src/main/java/com/example/aspect/LoggingAspect.java (Unchanged)

```
package com.example.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Arrays;

@Aspect
public class LoggingAspect {
    private static final Logger logger = LoggerFactory.getLogger(LoggingAspect.class);

    @Around("@annotation(com.example.annotation.Loggable)")
    public Object logMethod(ProceedingJoinPoint joinPoint) throws Throwable {
        String methodName = joinPoint.getSignature().getName();
        String className = joinPoint.getTarget().getClass().getSimpleName();
        Object[] args = joinPoint.getArgs();

        logger.trace("Entering {}.{} with arguments: {}", className, methodName,
        Arrays.toString(args));

        try {
            Object result = joinPoint.proceed();
            logger.info("Exiting {}.{} with result: {}", className, methodName, result);
            return result;
        } catch (Throwable e) {
            logger.error("Exception in {}.{}: {}", className, methodName, e.getMessage(), e);
            throw e;
        }
    }
}
```

6. src/main/java/com/example/model/User.java (Unchanged)

```
package com.example.model;

public class User {
    private final String id;
    private final String name;

    public User(String id, String name) {
        this.id = id;
        this.name = name;
    }
}
```



```
public String getId() {
    return id;
}

public String getName() {
    return name;
}

@Override
public String toString() {
    return "User{id='" + id + "', name='" + name + "'}";
}
}
```

7. src/main/java/com/example/exception/CustomException.java (Unchanged)

```
package com.example.exception;

public class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }

    public CustomException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

8. src/main/java/com/example/service/UserService.java (Remove @Service)

```
package com.example.service;

import com.example.annotation.Loggable;
import com.example.exception.CustomException;
import com.example.model.User;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class UserService {
    private static final Logger logger = LoggerFactory.getLogger(UserService.class);

    @Loggable
    public User processUser(String userId, String name) throws CustomException {
        if (userId == null || userId.trim().isEmpty()) {
            logger.warn("Invalid userId provided: {}", userId);
            throw new CustomException("User ID cannot be null or empty");
        }
    }
}
```



```
    logger.debug("Creating user object for userId: {}", userId);
    User user = new User(userId, name);
    return user;
}
}
```

9. src/main/java/com/example/Main.java (Unchanged)

```
package com.example;

import com.example.config.AppConfig;
import com.example.exception.CustomException;
import com.example.model.User;
import com.example.service.UserService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
    private static final Logger logger = LoggerFactory.getLogger(Main.class);

    public static void main(String[] args) {
        logger.info("Starting application");

        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        UserService userService = context.getBean(UserService.class);

        try {
            // Successful case
            User user = userService.processUser("123", "John Doe");
            logger.info("Created user: {} with name: {}", user.getId(), user.getName());

            // Error case
            userService.processUser(null, "Jane Doe");
        } catch (CustomException e) {
            logger.error("Application error occurred", e);
        }

        logger.info("Application shutdown");
    }
}
```

How @Loggable Works

In the provided program, the `@Loggable` annotation is a custom annotation used to enable automatic logging of method entry, exit, and exceptions using Aspect-Oriented Programming (AOP) with Spring's AOP framework (`spring-aop`). It works by intercepting methods annotated with



@Loggable and adding logging behavior without modifying the core business logic. Below is a concise explanation of how @Loggable works in the program, focusing on its implementation and interaction with the logging system.

Definition of @Loggable Annotation:

- Located in src/main/java/com/example/annotation/Loggable.java:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Loggable {
}
```

- @Retention(RetentionPolicy.RUNTIME) ensures the annotation is available at runtime for AOP processing.
- @Target(ElementType.METHOD) restricts the annotation to methods only.

Usage in UserService:

- The processUser method in UserService is annotated with @Loggable:

```
@Loggable
public User processUser(String userId, String name) throws CustomException {
    // Business logic
}
```

- This marks the method for interception by the AOP aspect.

AOP Aspect (LoggingAspect):

- Defined in src/main/java/com/example/aspect/LoggingAspect.java:

```
@Aspect
public class LoggingAspect {
    private static final Logger logger = LoggerFactory.getLogger(LoggingAspect.class);

    @Around("@annotation(com.example.annotation.Loggable)")
    public Object logMethod(ProceedingJoinPoint joinPoint) throws Throwable {
        String methodName = joinPoint.getSignature().getName();
        String className = joinPoint.getTarget().getClass().getSimpleName();
        Object[] args = joinPoint.getArgs();

        logger.trace("Entering {}.{} with arguments: {}", className, methodName, Arrays.toString(args));

        try {
            Object result = joinPoint.proceed();
            logger.info("Exiting {}.{} with result: {}", className, methodName, result);
            return result;
        } catch (Throwable e) {
            logger.error("Exception in {}.{}: {}", className, methodName, e.getMessage(), e);
            throw e;
        }
    }
}
```



Rewrite above Student application to demonstrate the use of AOP. Using AOP, generate the log which will give the idea about the execution track of the application.

The console-based CRUD application (using MySQL, Spring Core, JdbcTemplate, and annotation-based configuration) to incorporate Aspect-Oriented Programming (AOP) using Spring AOP. The goal is to add logging to track the execution flow of the application, specifically for key methods in the StudentController and StudentDAOImpl classes. This will provide insights into method calls, their arguments, return values, and any exceptions, without modifying the core business logic. The logging will be implemented using an AOP aspect to intercept method executions and log relevant details.

Approach

- **AOP Concept:** Use Spring AOP to create an aspect that logs method entry, exit, and exceptions for the application's key components.
- **Logging:** Log method names, arguments, return values, and exceptions using SLF4J with Logback as the logging framework.
- **Annotations:** Define a custom annotation (e.g., @Loggable) to mark methods for logging, making the aspect reusable and selective.
- **Components:** Keep the existing MVC structure, MySQL database, JdbcTemplate, and annotation-based configuration.
- **RowMapper:** Retain the non-lambda RowMapper (anonymous inner class) from the previous version.

1. Maven Dependencies (pom.xml)

Add dependencies for Spring AOP and SLF4J with Logback for logging, alongside existing MySQL and Spring dependencies.

pom.xml:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>student-crud</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>

  <dependencies>
    <!-- MySQL Connector -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.33</version>
    </dependency>
  </dependencies>

```



```

<!-- Spring Core and JDBC -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.39</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.3.39</version>
</dependency>
<!-- Spring AOP -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.3.39</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.22</version>
</dependency>
<!-- SLF4J and Logback for Logging -->
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.4.14</version>
</dependency>
</dependencies>
</project>

```

Changes:

- Added spring-aop and aspectjweaver for AOP support.
- Added logback-classic (includes SLF4J) for logging.

2. Logging Configuration (logback.xml)

Create a logback.xml file to configure logging output to the console with a clear format.

src/main/resources/logback.xml:

```

<configuration>
    <!-- Console Appender -->
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level [%logger{36}] - %msg%n</pattern>
        </encoder>
    </appender>

    <!-- File Appender -->
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>logs/student-crud.log</file>

```



```

<append>true</append>
<encoder>
    <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level [%logger{36}] - %msg%n</pattern>
</encoder>
</appender>

<!-- Logger for com.example package -->
<logger name="com.example" level="DEBUG" additivity="false">
    <appender-ref ref="CONSOLE"/>
    <appender-ref ref="FILE"/>
</logger>

<!-- Root Logger -->
<root level="INFO">
    <appender-ref ref="CONSOLE"/>
    <appender-ref ref="FILE"/>
</root>
</configuration>

```

Explanation:

- Configures Logback to log messages at DEBUG level for the com.example package.
- Logs include timestamp, log level, logger name, and message.
- Outputs to the console for simplicity.

3. Custom Logging Annotation (Loggable.java)

Create a custom annotation to mark methods for logging, allowing selective application of the AOP aspect.

src/main/java/com/example/annotation/Loggable.java:

```

package com.example.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

```

```

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Loggable {
}

```

Explanation:

- @Loggable will be used to annotate methods in StudentController and StudentDAOImpl that we want to log.
- Retention is RUNTIME to allow AOP to detect the annotation.
- Target is METHOD to restrict usage to methods.

4. AOP Logging Aspect (LoggingAspect.java)

Create an aspect to log method execution details (entry, exit, exceptions) for methods annotated with @Loggable.

src/main/java/com/example/aspect/LoggingAspect.java:



```
package com.example.aspect;

import com.example.annotation.Loggable;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Aspect;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import java.util.Arrays;

@Aspect
@Component
public class LoggingAspect {
    private static final Logger logger = LoggerFactory.getLogger(LoggingAspect.class);

    @Before("@annotation(com.example.annotation.Loggable)")
    public void logBefore(JoinPoint joinPoint) {
        String methodName = joinPoint.getSignature().toShortString();
        Object[] args = joinPoint.getArgs();
        logger.debug("Entering method: {} with arguments: {}", methodName, Arrays.toString(args));
    }

    @AfterReturning(pointcut = "@annotation(com.example.annotation.Loggable)", returning =
"result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {
        String methodName = joinPoint.getSignature().toShortString();
        logger.debug("Exiting method: {} with return value: {}", methodName, result);
    }

    @AfterThrowing(pointcut = "@annotation(com.example.annotation.Loggable)", throwing =
"exception")
    public void logAfterThrowing(JoinPoint joinPoint, Throwable exception) {
        String methodName = joinPoint.getSignature().toShortString();
        logger.error("Exception in method: {} with message: {}", methodName, exception.getMessage());
    }
}
```

Explanation:

- @Aspect and @Component mark this as a Spring-managed aspect.
- @Before: Logs method entry with method name and arguments before execution.
- @AfterReturning: Logs method exit with the return value after successful execution.
- @AfterThrowing: Logs any exceptions thrown during method execution.
- Uses @Loggable as the pointcut to target annotated methods.



- SLF4J with Logback logs messages at DEBUG (entry/exit) and ERROR (exceptions) levels.

5. Spring Configuration (SpringConfig.java)

Enable AOP with @EnableAspectJAutoProxy and keep component scanning.

src/main/java/com/example/config/SpringConfig.java:

```
package com.example.config;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import javax.sql.DataSource;

@Configuration
@ComponentScan(basePackages = "com.example")
@EnableAspectJAutoProxy
public class SpringConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");

        dataSource.setUrl("jdbc:mysql://localhost:3306/studentdb?useSSL=false&serverTimezone=UTC");
        dataSource.setUsername("root"); // Replace with your MySQL username
        dataSource.setPassword("your_password"); // Replace with your MySQL password
        return dataSource;
    }

    @Bean
    public JdbcTemplate jdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }
}
```

Changes:

- Added @EnableAspectJAutoProxy to enable Spring AOP support.
- No changes to DataSource or JdbcTemplate beans.

6. Database Initializer (DatabaseInitializer.java)

No changes needed, as it already uses @Component and @PostConstruct.

src/main/java/com/example/config/DatabaseInitializer.java:

```
package com.example.config;
```

```
import org.springframework.beans.factory.annotation.Autowired;
```



```
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;

@Component
public class DatabaseInitializer {
    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public DatabaseInitializer(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @PostConstruct
    public void initialize() {
        try {
            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
                )
                """;
            jdbcTemplate.execute(sql);
            System.out.println("Table 'student' created or already exists.");
        } catch (Exception e) {
            System.err.println("Error initializing database: " + e.getMessage());
            throw new RuntimeException("Failed to initialize database", e);
        }
    }
}
```

7. Student Model (Student.java)

Unchanged, as it's a POJO.

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 + "'}";
}
}
```

8. Student DAO Interface (StudentDAO.java)

Unchanged.

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();
}
```

9. Student DAO Implementation (StudentDAOImpl.java)

Add @Loggable annotations to DAO methods to enable logging via the AOP aspect.

src/main/java/com/example/dao/StudentDAOImpl.java:

```
package com.example.dao;
```

```
import com.example.annotation.Loggable;
import com.example.model.Student;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
```



```
@Repository
public class StudentDAOImpl implements StudentDAO {
    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public StudentDAOImpl(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    private final RowMapper<Student> studentRowMapper = new RowMapper<Student>() {
        @Override
        public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
            return new Student(
                rs.getInt("id"),
                rs.getString("name"),
                rs.getString("email")
            );
        }
    };

    @Override
    @Loggable
    public void create(Student student) {
        String sql = "INSERT INTO student (name, email) VALUES (?, ?)";
        try {
            jdbcTemplate.update(sql, student.getName(), student.getEmail());
            System.out.println("Student created successfully in DAO.");
        } catch (Exception e) {
            System.err.println("Error creating student: " + e.getMessage());
            throw new RuntimeException("Failed to create student", e);
        }
    }

    @Override
    @Loggable
    public Student read(int id) {
        String sql = "SELECT * FROM student WHERE id = ?";
        try {
            return jdbcTemplate.queryForObject(sql, new Object[]{id}, studentRowMapper);
        } catch (Exception e) {
            System.err.println("Error reading student: " + e.getMessage());
            return null;
        }
    }

    @Override
    @Loggable
```



```

public void update(Student student) {
    String sql = "UPDATE student SET name = ?, email = ? WHERE id = ?";
    try {
        jdbcTemplate.update(sql, student.getName(), student.getEmail(), student.getId());
        System.out.println("Student updated successfully in DAO.");
    } catch (Exception e) {
        System.err.println("Error updating student: " + e.getMessage());
        throw new RuntimeException("Failed to update student", e);
    }
}

@Override
@Loggable
public void delete(int id) {
    String sql = "DELETE FROM student WHERE id = ?";
    try {
        jdbcTemplate.update(sql, id);
        System.out.println("Student deleted successfully in DAO.");
    } catch (Exception e) {
        System.err.println("Error deleting student: " + e.getMessage());
        throw new RuntimeException("Failed to delete student", e);
    }
}

@Override
@Loggable
public List<Student> getAll() {
    String sql = "SELECT * FROM student";
    try {
        return jdbcTemplate.query(sql, studentRowMapper);
    } catch (Exception e) {
        System.err.println("Error retrieving students: " + e.getMessage());
        throw new RuntimeException("Failed to retrieve students", e);
    }
}
}

```

Changes:

- Added @Loggable to all DAO methods to enable AOP logging.
- No changes to JdbcTemplate logic or RowMapper (still using anonymous inner class).

10. Student Controller (StudentController.java)

Add @Loggable annotations to controller methods.

src/main/java/com/example/controller/StudentController.java:

```
package com.example.controller;
```

```
import com.example.annotation.Loggable;
import com.example.dao.StudentDAO;
import com.example.model.Student;
```



```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class StudentController {
    private final StudentDAO studentDAO;

    @Autowired
    public StudentController(StudentDAO studentDAO) {
        this.studentDAO = studentDAO;
    }

    @Loggable
    public void createStudent(String name, String email) {
        Student student = new Student(0, name, email);
        studentDAO.create(student);
        System.out.println("Student created successfully");
    }

    @Loggable
    public Student getStudent(int id) {
        Student student = studentDAO.read(id);
        if (student == null) {
            System.out.println("Student not found");
        }
        return student;
    }

    @Loggable
    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");
    }

    @Loggable
    public void deleteStudent(int id) {
        studentDAO.delete(id);
        System.out.println("Student deleted successfully");
    }

    @Loggable
    public List<Student> getAllStudents() {
        return studentDAO.getAll();
    }
}
```



Changes:

- Added @Loggable to all public methods to enable AOP logging.

11. Student View (StudentView.java)

No changes needed, as we won't apply AOP logging to the view layer to keep console output clean for users.

src/main/java/com/example/view/StudentView.java:

package com.example.view;

```
import com.example.controller.StudentController;
import com.example.model.Student;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.util.Scanner;

@Component
public class StudentView {
    private final StudentController controller;
    private final Scanner scanner;

    @Autowired
    public StudentView(StudentController controller) {
        this.controller = controller;
        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 -> {
                    break;
                }
            }
        }
    }

    private void createStudent() {
        System.out.print("Enter Student ID: ");
        String id = scanner.nextLine();
        System.out.print("Enter Student Name: ");
        String name = scanner.nextLine();
        System.out.print("Enter Student Age: ");
        int age = scanner.nextInt();
        System.out.print("Enter Student Marks: ");
        int marks = scanner.nextInt();

        Student student = new Student(id, name, age, marks);
        controller.createStudent(student);
        System.out.println("Student created successfully!");
    }

    private void viewStudent() {
        System.out.print("Enter Student ID: ");
        String id = scanner.nextLine();
        Student student = controller.viewStudent(id);
        if (student != null) {
            System.out.println("Student found: " + student);
        } else {
            System.out.println("Student not found.");
        }
    }

    private void updateStudent() {
        System.out.print("Enter Student ID: ");
        String id = scanner.nextLine();
        System.out.print("Enter New Name: ");
        String name = scanner.nextLine();
        System.out.print("Enter New Age: ");
        int age = scanner.nextInt();
        System.out.print("Enter New Marks: ");
        int marks = scanner.nextInt();

        Student student = new Student(id, name, age, marks);
        controller.updateStudent(student);
        System.out.println("Student updated successfully!");
    }

    private void deleteStudent() {
        System.out.print("Enter Student ID: ");
        String id = scanner.nextLine();
        controller.deleteStudent(id);
        System.out.println("Student deleted successfully!");
    }

    private void listAllStudents() {
        List<Student> students = controller.listAllStudents();
        if (!students.isEmpty()) {
            for (Student student : students) {
                System.out.println(student);
            }
        } else {
            System.out.println("No students found.");
        }
    }
}
```



```
        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);  
    }  
}
```



```
}
```

12. Main Class (Main.java)

Unchanged.

`src/main/java/com/example/Main.java:`

```
package com.example;

import com.example.config.SpringConfig;
import com.example.view.StudentView;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
    public static void main(String[] args) {
        // Initialize Spring context
        AnnotationConfigApplicationContext context = new
        AnnotationConfigApplicationContext(SpringConfig.class);

        // Get StudentView bean and start the application
        StudentView view = context.getBean(StudentView.class);
        view.displayMenu();

        // Close context when done
        context.close();
    }
}
```

Step-by-Step Explanation of Logging Process

Let's assume a user selects option 1 from the console menu to create a student with name "John Doe" and email "john.doe@example.com". Here's how logging occurs:

Step 1: User Interaction Triggers the Operation

- **Action:** The user runs the application (`mvn exec:java -Dexec.mainClass="com.example.Main"`) and interacts with the console menu in `StudentView`.
- **Code:** In `StudentView`, the `createStudent` method is called:

```
private void createStudent() {
    System.out.print("Enter name: ");
    String name = scanner.nextLine(); // User enters "John Doe"
    System.out.print("Enter email: ");
    String email = scanner.nextLine(); // User enters "john.doe@example.com"
    controller.createStudent(name, email);
}

• Outcome: StudentController.createStudent("John Doe", "john.doe@example.com") is invoked.
```

Step 2: AOP Intercepts `StudentController.createStudent`

- **Trigger:** The `createStudent` method in `StudentController` is annotated with `@Loggable`:



```

@Loggable
public void createStudent(String name, String email) {
    Student student = new Student(0, name, email);
    studentDAO.create(student);
    System.out.println("Student created successfully");
}

```

- **AOP Aspect:** The LoggingAspect intercepts this method because it matches the @Loggable pointcut:

```

@Before("@annotation(com.example.annotation.Loggable)")
public void logBefore(JoinPoint joinPoint) {
    String methodName = joinPoint.getSignature().toShortString();
    Object[] args = joinPoint.getArgs();
    logger.debug("Entering method: {} with arguments: {}", methodName, Arrays.toString(args));
}

```

- **Execution:**

- JoinPoint provides method metadata: methodName is StudentController.createStudent(..), and args is ["John Doe", "john.doe@example.com"].
- The SLF4J logger writes a DEBUG message: "Entering method: StudentController.createStudent(..) with arguments: [John Doe, john.doe@example.com]".

- **Log Output:**

- **Console** (via CONSOLE appender):

2025-06-19 15:09:00 DEBUG [com.example.aspect.LoggingAspect] - Entering method: StudentController.createStudent(..) with arguments: [John Doe, john.doe@example.com]

- **File** (logs/student-crud.log, via FILE appender):

2025-06-19 15:09:00 DEBUG [com.example.aspect.LoggingAspect] - Entering method: StudentController.createStudent(..) with arguments: [John Doe, john.doe@example.com]

Creating Web-Application using Spring-Core:

Web application using Spring Core (Spring MVC), JSP for the UI, HikariCP as the DataSource, and MySQL as the database. The application will include an index page as the starting point, a separate page to view logs, and CSS for styling the UI.

What is MVC, Spring MVC?

MVC, or Model-View-Controller, is a design pattern commonly used in software development, particularly for web applications and user interfaces. It divides application logic into three interconnected components:

- **Model:** Represents the data, business logic, and rules of the application. It handles the underlying structure and storage, managing the data and responding to requests from the Controller.
- **View:** The user interface, displaying the data from the Model to the user. It presents the data in a visual format and updates when the Model changes.



- **Controller:** Acts as an intermediary between the Model and View. It processes user inputs, communicates with the Model to update data, and ensures the View reflects those changes.

This separation promotes organized code, modularity, and easier maintenance by keeping concerns distinct. For example, in a web app, the Model might handle database queries, the View renders HTML pages, and the Controller processes user clicks or form submissions.

What is Spring MVC?

Spring MVC (Model-View-Controller) is a module of the Spring Framework designed for building web applications and RESTful APIs. It leverages **Spring Core**'s dependency injection and inversion of control (IoC) to provide a robust framework for handling HTTP requests, rendering views, and managing web application workflows. Spring MVC follows the MVC architectural pattern, separating concerns into three components:

- **Model:** Represents the data or business logic (e.g., Java objects or database entities).
- **View:** Handles the user interface (e.g., HTML, JSP, Thymeleaf templates, or JSON for APIs).
- **Controller:** Manages user requests, processes input, interacts with the model, and selects the view to render.

Spring MVC is widely used for its flexibility, scalability, and integration with other Spring modules (e.g., Spring Data, Spring Security).

How Does Spring MVC Work?

Spring MVC processes HTTP requests through a well-defined workflow, centered around the **DispatcherServlet**, the core component that acts as a front controller. Here's a step-by-step explanation of how it works:

1. Client Sends HTTP Request

A user sends an HTTP request (e.g., GET /hello) via a browser or API client to the Spring MVC application.

2. DispatcherServlet Receives the Request

The **DispatcherServlet**, configured in the web application, intercepts all incoming requests. It is the central servlet that coordinates the request processing workflow.

- Configured in web.xml (traditional) or via Spring Boot auto-configuration.
- Acts as the front controller, delegating tasks to other components.

3. Handler Mapping

The DispatcherServlet consults **Handler Mappings** to determine which **Controller** and method should handle the request based on the URL, HTTP method, or annotations (e.g., @GetMapping("/hello")).

- Example: A @RequestMapping annotation on a controller method maps the request to a specific handler.

4. Controller Processes the Request

The selected **Controller** method executes, performing the following:

- Processes input (e.g., query parameters, form data, or JSON payload).
- Interacts with the **Model** (e.g., services or repositories) to fetch or update data.
- Prepares data for the response (e.g., adds attributes to the model for views or returns a response body for APIs).



Example Controller:

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class HelloController {
    @GetMapping("/hello")
    public String sayHello(Model model) {
        model.addAttribute("message", "Hello, Spring MVC!");
        return "hello"; // View name
    }
}
```

5. View Resolution

If the controller returns a view name (e.g., "hello"), the DispatcherServlet uses a **View Resolver** to map the logical view name to a physical view (e.g., a Thymeleaf template, JSP, or JSON for REST).

- For REST APIs, `@RestController` or `@ResponseBody` skips view resolution and serializes the return value (e.g., Java object to JSON).
- Example View (Thymeleaf hello.html):

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Hello</title>
</head>
<body>
    <h1 th:text="${message}">Default Message</h1>
</body>
</html>
```

6. Response Rendering

The resolved view renders the response (e.g., HTML for a web page or JSON for an API) and sends it back to the client via the DispatcherServlet.

7. Exception Handling (Optional)

If an error occurs, Spring MVC uses **Exception Handlers** (e.g., `@ExceptionHandler` or `@ControllerAdvice`) to handle exceptions and return appropriate responses.

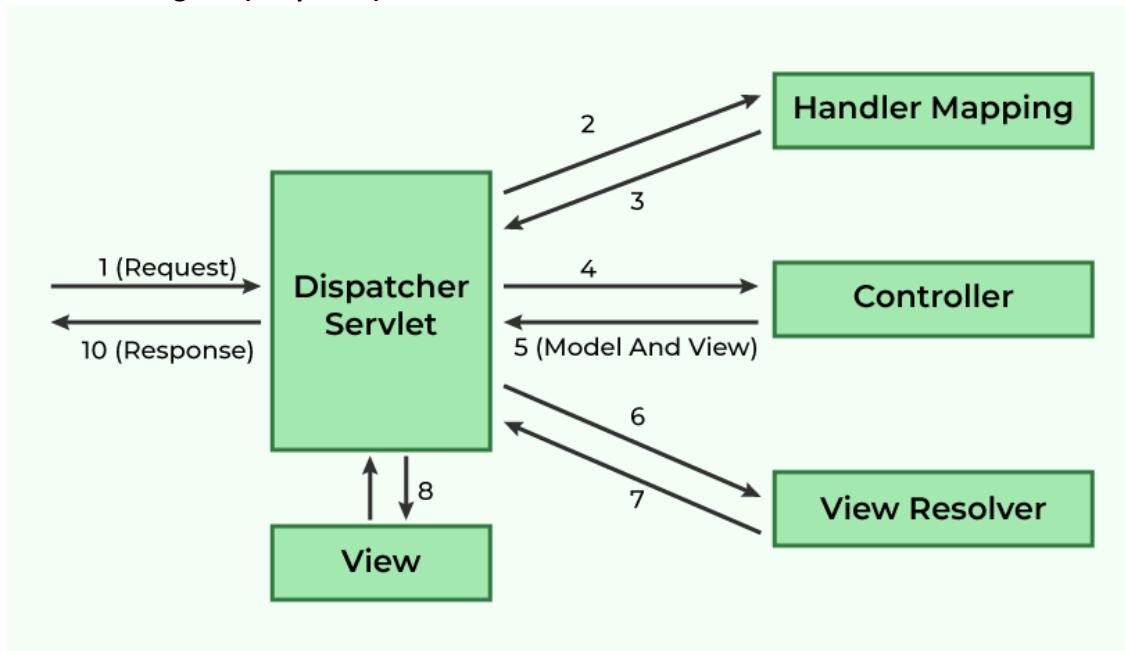
Key Components of Spring MVC

- **DispatcherServlet**: Central servlet that dispatches requests to handlers.
- **HandlerMapping**: Maps requests to controller methods.
- **Controller**: Processes requests and prepares the model or response.
- **ViewResolver**: Resolves view names to actual view templates.
- **Model**: Holds data to be rendered in the view.
- **View**: Renders the final output (e.g., HTML, JSON).



- **HandlerInterceptor**: Allows pre- and post-processing of requests (e.g., for logging or authentication).
- **HandlerExceptionResolver**: Manages exceptions during request processing.

Workflow Diagram (Simplified)



Example:

Spring Core-based web application using Spring MVC, including the project structure and necessary code. The application will include a simple endpoint and a view,

Step-by-Step Guide

1. Set Up the Project

Use Maven to create a basic Java project. We'll avoid Spring Boot's starter dependencies to focus on Spring Core and Spring MVC.

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>spring-mvc-demo</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>spring-mvc-demo</name>
  <url>http://maven.apache.org</url>

  <properties>
  
```



```
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<spring.version>6.1.14</spring.version>
</properties>

<dependencies>
    <!-- Spring Core -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <!-- Spring MVC -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <!-- Servlet API (for web application) -->
    <dependency>
        <groupId>jakarta.servlet</groupId>
        <artifactId>jakarta.servlet-api</artifactId>
        <version>6.0.0</version>
        <scope>provided</scope>
    </dependency>
    <!-- Thymeleaf for views -->
    <dependency>
        <groupId>org.thymeleaf</groupId>
        <artifactId>thymeleaf-spring6</artifactId>
        <version>3.1.2.RELEASE</version>
    </dependency>
    <!-- Lombok (optional) -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.34</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
</project>
```

- **Key Dependencies:**

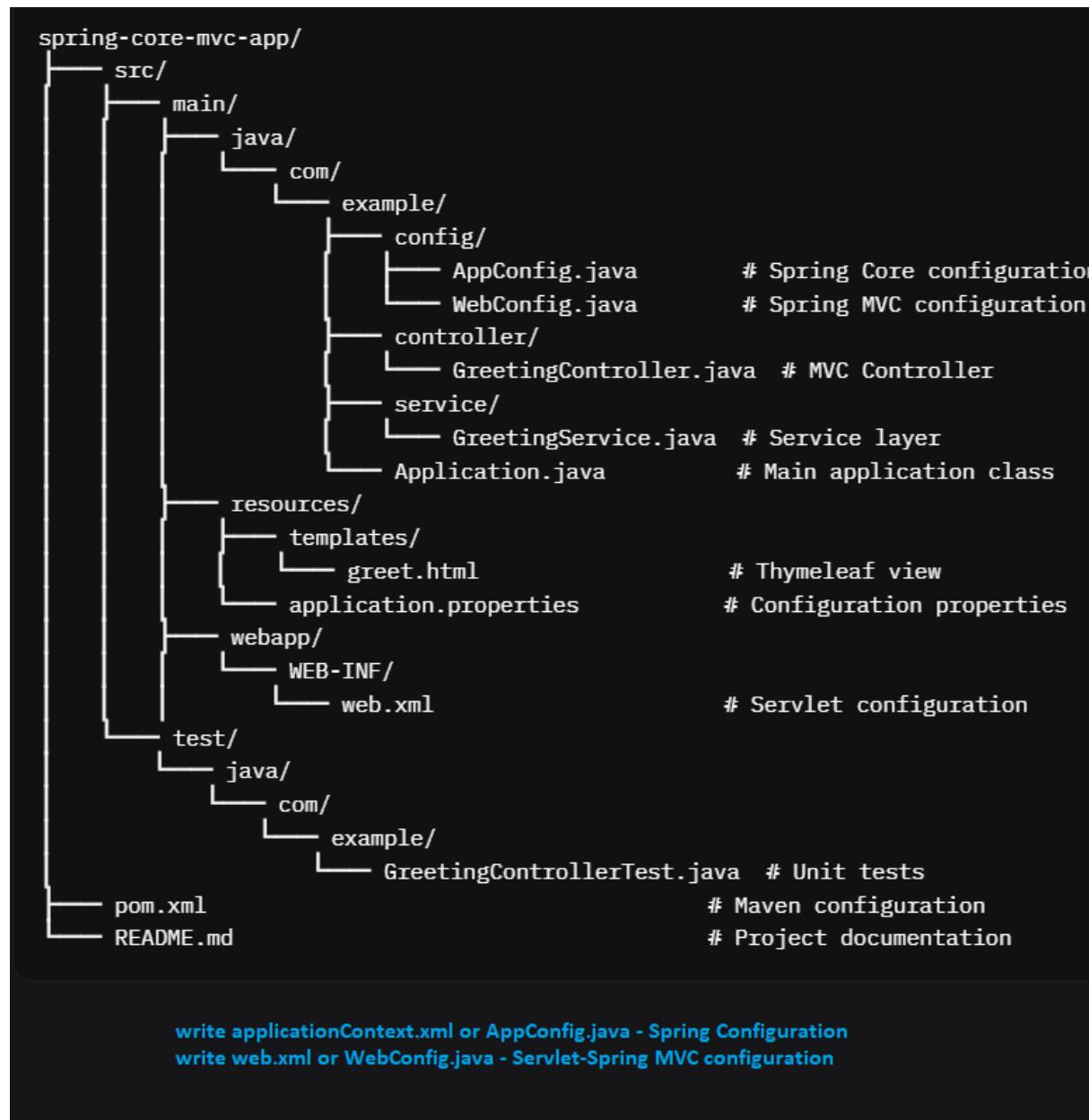
- spring-core and spring-context for IoC and dependency injection.
- spring-webmvc for Spring MVC functionality.



- jakarta.servlet-api for servlet support.
- thymeleaf-spring6 for rendering views.
- lombok to reduce boilerplate (optional).

2. Project Structure

Here's the project structure for the Spring Core + Spring MVC application:



3. Configure Spring Core (AppConfig.java)

Define beans for the application using Spring Core's Java-based configuration.

src/main/java/com/example/config/AppConfig.java:

```
package com.example.config;
```

```
import com.example.service.GreetingService;
```



```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
public class AppConfig {

    @Bean
    public GreetingService greetingService() {
        return new GreetingService();
    }
}
```

- **@Configuration:** Marks the class as a source of bean definitions.
- **@Bean:** Defines a bean managed by Spring's IoC container.

4. Configure Spring MVC (WebConfig.java)

Set up Spring MVC components, including view resolvers and component scanning.

src/main/java/com/example/config/WebConfig.java:

```
package com.example.config;
```

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ViewResolverRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.thymeleaf.spring6.SpringTemplateEngine;
import org.thymeleaf.spring6.view.ThymeleafViewResolver;
import org.thymeleaf.templatemode.TemplateMode;
import org.thymeleaf.templateresolver.ClassLoaderTemplateResolver;
```

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.example.controller")
public class WebConfig implements WebMvcConfigurer {

    @Bean
    public ClassLoaderTemplateResolver templateResolver() {
        ClassLoaderTemplateResolver resolver = new ClassLoaderTemplateResolver();
        resolver.setPrefix("templates/");
        resolver.setSuffix(".html");
        resolver.setTemplateMode(TemplateMode.HTML);
        resolver.setCharacterEncoding("UTF-8");
        return resolver;
    }

    @Bean
    public SpringTemplateEngine templateEngine() {
        SpringTemplateEngine engine = new SpringTemplateEngine();
        engine.setTemplateResolver(templateResolver());
```



```

    return engine;
}

@Bean
public ThymeleafViewResolver viewResolver() {
    ThymeleafViewResolver resolver = new ThymeleafViewResolver();
    resolver.setTemplateEngine(templateEngine());
    resolver.setCharacterEncoding("UTF-8");
    return resolver;
}

@Override
public void configureViewResolvers(ViewResolverRegistry registry) {
    registry.viewResolver(viewResolver());
}
}

```

- `@EnableWebMvc`: Enables Spring MVC.
- `@ComponentScan`: Scans for controllers in the specified package.
- `WebMvcConfigurer`: Configures MVC settings, such as view resolvers.
- `ThymeleafViewResolver`: Configures Thymeleaf for rendering HTML views.

5. Configure the Servlet (web.xml)

Define the DispatcherServlet and load Spring configurations.

`src/main/webapp/WEB-INF/web.xml:`

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
    version="6.0">

    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-
value>
        </init-param>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>com.example.config.AppConfig,com.example.config.WebConfig</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>

```

- `DispatcherServlet`: Central servlet for handling HTTP requests.
- `contextConfigLocation`: Specifies the configuration classes (AppConfig and WebConfig).



Optional/Alternative Java base configuration code to web.xml

```
package com.example.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;
import org.springframework.web.filter.CharacterEncodingFilter;
import javax.servlet.FilterRegistration;
import javax.servlet.ServletContext;

public class WebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null; // No root application context configuration
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[]{WebConfig.class}; // Use WebConfig.java for DispatcherServlet
    }

    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"}; // Map DispatcherServlet to "/"
    }

    @Override
    protected void customizeRegistration(ServletRegistration.Dynamic registration) {
        registration.setLoadOnStartup(1); // Equivalent to <load-on-startup>1</load-on-startup>
    }

    @Override
    protected FilterRegistration.Dynamic[] getServletFilters() {
        // Configure CharacterEncodingFilter
        CharacterEncodingFilter encodingFilter = new CharacterEncodingFilter();
        encodingFilter.setEncoding("UTF-8");
        encodingFilter.setForceEncoding(true);

        FilterRegistration.Dynamic filterRegistration = getServletContext()
            .addFilter("encodingFilter", encodingFilter);
        filterRegistration.addMappingForUrlPatterns(null, false, "/*");

        return new FilterRegistration.Dynamic[]{filterRegistration};
    }
}
```

6. Create the Service Layer

Define a simple service to handle business logic.

src/main/java/com/example/service/GreetingService.java:

```
package com.example.service;
```

```
public class GreetingService {
    public String getGreeting(String name) {
        return "Hello, " + name + "!";
    }
}
```



7. Create the Controller

Define a controller to handle HTTP requests and render views.

src/main/java/com/example/controller/GreetingController.java:

```
package com.example.controller;

import com.example.service.GreetingService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class GreetingController {

    private final GreetingService greetingService;

    @Autowired
    public GreetingController(GreetingService greetingService) {
        this.greetingService = greetingService;
    }

    @GetMapping("/greet")
    public String greet(@RequestParam(name = "name", defaultValue = "World") String name,
    Model model) {
        model.addAttribute("name", greetingService.getGreeting(name));
        return "greet"; // Maps to greet.html
    }
}
```

- `@Controller`: Marks the class as a Spring MVC controller.
- `@Autowired`: Injects the `GreetingService` bean (managed by Spring Core).
- `@GetMapping`: Maps GET requests to `/greet`.

8. Create the View

Create a Thymeleaf template for rendering the response.

src/main/resources/templates/greet.html:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Greeting</title>
</head>
<body>
    <h1 th:text="${name}">Hello</h1>
</body>
</html>
```

9. Application Properties (Optional)



Add any additional configurations in application.properties.

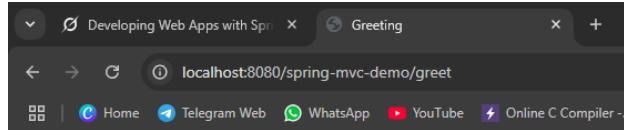
src/main/resources/application.properties:

```
# Optional: Add custom properties if needed
server.port=8080
```

10. Run Application

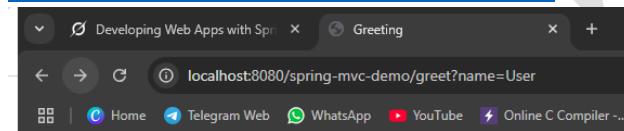
- deployed to a servlet container.
- Open Browser and Test

- <http://localhost:8080/spring-mvc-demo/greet?name=User>



Hello, World!

- <http://localhost:8080/spring-mvc-demo/greet>



Hello, User!

* We can use XML Configuration(Optional/Alternative) rather than Class Configuration- WebConfig.java, for above application we can use the XML Configuration as,

Equivalent web.xml

The web.xml file configures the Spring DispatcherServlet to handle requests and specifies the location of the Spring configuration file (dispatcher-servlet.xml).

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
          version="4.0">

    <!-- Define the Spring DispatcherServlet -->
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- Specify the Spring configuration file -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/dispatcher-servlet.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
```



```

<!-- Map the DispatcherServlet to handle all requests -->
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>

<!-- Character Encoding Filter for UTF-8 -->
<filter>
  <filter-name>encodingFilter</filter-name>
  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
  <init-param>
    <param-name>forceEncoding</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>encodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

</web-app>

```

Equivalent dispatcher-servlet.xml

The dispatcher-servlet.xml file replicates the Spring MVC and Thymeleaf configuration from WebConfig.java. It should be placed in the WEB-INF directory of the web application.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd
           http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd">

  <!-- Enable component scanning for controllers -->
  <context:component-scan base-package="com.example.controller"/>

  <!-- Enable Spring MVC annotations -->
  <mvc:annotation-driven/>

  <!-- Thymeleaf Template Resolver -->
  <bean id="templateResolver" class="org.thymeleaf.templateresolver.ClassLoaderTemplateResolver">
    <property name="prefix" value="templates/" />
    <property name="suffix" value=".html" />
    <property name="templateMode" value="HTML" />
    <property name="characterEncoding" value="UTF-8" />
  </bean>

  <!-- Thymeleaf Template Engine -->
  <bean id="templateEngine" class="org.thymeleaf.spring6.SpringTemplateEngine">
    <property name="templateResolver" ref="templateResolver" />
  </bean>

  <!-- Thymeleaf View Resolver -->
  <bean id="viewResolver" class="org.thymeleaf.spring6.view.ThymeleafViewResolver">

```



```
<property name="templateEngine" ref="templateEngine"/>
<property name="characterEncoding" value="UTF-8"/>
</bean>

</beans>
```

Now lets Above student application to web application

Project Description: Student CRUD Web Application

The **Student CRUD Web Application** is a Java-based web application designed to manage student records using a Create, Read, Update, and Delete (CRUD) interface. Built with Spring MVC, it provides a user-friendly JSP-based UI for performing operations on student data stored in a MySQL database. The application incorporates HikariCP for efficient database connection pooling, Logback for logging, and AOP for cross-cutting concerns like logging method execution. It leverages Jakarta EE 10 (Servlet 6.0) and is deployed on Apache Tomcat 10.1, ensuring modern web standards and compatibility.

Key Features

- **Student Management:**
 - Create, view, edit, and delete student records (ID, name, email).
 - Validation ensures non-empty names, valid email formats, and unique emails.
- **Web Interface:**
 - JSP pages (list.jsp, create.jsp, edit.jsp, view.jsp, logs.jsp, index.jsp) with JSTL for dynamic rendering.
 - CSS styling for a clean, responsive UI.
- **Database Integration:**
 - MySQL database (studentdb) with a student table.
 - HikariCP for optimized connection management.
 - Spring JDBC for database operations via StudentDAO.
- **Logging:**
 - Logback writes logs to student-crud-web/logs/student-crud.log in the project directory.
 - LogViewer displays logs via the /students/logs endpoint.
 - AOP-based logging tracks method entry/exit and exceptions.
- **Error Handling:**
 - Graceful handling of database and file access errors.
 - User-friendly error messages in the UI.

Technology Stack

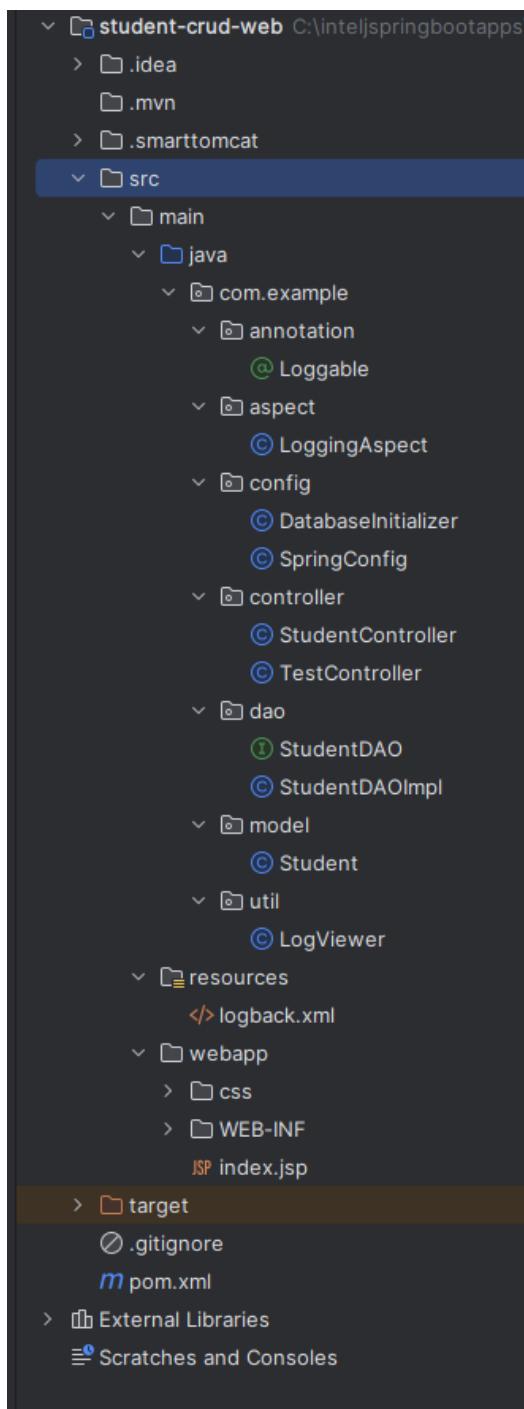
- **Backend:** Spring MVC 6.1.14, Spring JDBC, Spring AOP
- **Frontend:** JSP, JSTL (Jakarta EE 10), CSS
- **Database:** MySQL 8.0, HikariCP 5.1.0
- **Logging:** Logback 1.5.13, SLF4J
- **Server:** Apache Tomcat 10.1 (Servlet 6.0, Jakarta EE 10)
- **Build Tool:** Maven
- **Java:** 17
- **Validation:** Jakarta Validation 3.0.2, Hibernate Validator 8.0.1

Project Structure

- **Source Code:** src/main/java/com/example/ (controller, dao, model, util, config, aspect, annotation)
- **Resources:** src/main/resources/ (logback.xml)



- **Web Assets:** src/main/webapp/ (WEB-INF/views/*.jsp, css/, web.xml)
- **Logs:** student-crud-web/logs/student-crud.log
- **WAR:** target/student-crud-web.war



Challenges Addressed:

1. HTTP 404 Error (/students Endpoint)

- **Cause:**
 - Incompatible web.xml (Servlet 6.0, Jakarta EE 10) with Tomcat version or dependencies.



- Mixed javax and jakarta dependencies in pom.xml.
- Missing <welcome-file-list> in web.xml.
- **Solution:**
 - Updated web.xml to ensure Servlet 6.0 compatibility and added <welcome-file-list> for index.jsp.
 - Revised pom.xml to use jakarta.servlet-api:6.0.0, jakarta.servlet.jsp.jstl:3.0.1, and compatible validation APIs.
 - Ensured Tomcat 10.1+ for Jakarta EE 10 support.
 - Verified SpringConfig.java for correct @ComponentScan and @EnableWebMvc.

2. HTTP 500 Error (/students Endpoint)

- **Cause:**
 - Potential database connection failure in StudentDAOImpl.getAll().
 - JSP rendering issues due to JSTL or EL errors.
- **Solution:**
 - Validated MySQL connection and studentdb schema in SpringConfig.java.
 - Updated JSPs to use jakarta.tags.core for JSTL compatibility.
 - Added try-catch in StudentController.listStudents to handle exceptions gracefully.

3. Log Reading Error (/students/logs Endpoint)

- **Cause:**
 - Added SLF4J dependency
- **Solution:**
 - Updated logback.xml to write to student-crud-web/log/student-crud.log using \${user.dir}/log.
 - Fixed LogViewer.java path to System.getProperty("user.dir") + "/log/student-crud.log" and removed semicolon.
 - Ensured LogViewer handles exceptions with fallback error messages.

Project Source Code:

- **pom.xml:** Defines Maven dependencies and build configuration for Spring 6.1.14, Jakarta EE 10, and MySQL.
- ```

<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.example</groupId>
 <artifactId>student-crud-web</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>war</packaging>

 <properties>
 <maven.compiler.source>11</maven.compiler.source>
 <maven.compiler.target>11</maven.compiler.target>
 <spring.version>6.1.14</spring.version>
 <slf4j.version>2.0.16</slf4j.version>
 <logback.version>1.5.12</logback.version>
 </properties>

 <dependencies>
 <!-- Spring Core -->
 <dependency>
 <groupId>org.springframework</groupId>

```



```
<artifactId>spring-core</artifactId>
<version>${spring.version}</version>
</dependency>
<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-context</artifactId>
 <version>${spring.version}</version>
</dependency>
<!-- Spring MVC -->
<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-webmvc</artifactId>
 <version>${spring.version}</version>
</dependency>
<!-- Spring JDBC -->
<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-jdbc</artifactId>
 <version>${spring.version}</version>
</dependency>

<!-- Servlet API (for web application) -->
<dependency>
 <groupId>jakarta.servlet</groupId>
 <artifactId>jakarta.servlet-api</artifactId>
 <version>6.0.0</version>
 <scope>provided</scope>
</dependency>
<!-- MySQL Connector -->
<dependency>
 <groupId>mysql</groupId>
 <artifactId>mysql-connector-java</artifactId>
 <version>8.0.33</version>
</dependency>
<!-- HikariCP -->
<dependency>
 <groupId>com.zaxxer</groupId>
 <artifactId>HikariCP</artifactId>
 <version>5.1.0</version>
</dependency>
<!-- JSTL for JSP -->
<dependency>
 <groupId>jakarta.servlet.jsp.jstl</groupId>
 <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
 <version>3.0.0</version>
</dependency>
<dependency>
 <groupId>org.glassfish.web</groupId>
 <artifactId>jakarta.servlet.jsp.jstl</artifactId>
 <version>3.0.1</version>
</dependency>
<!-- SLF4J and Logback -->
<!-- Spring AOP -->
<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-aop</artifactId>
 <version>${spring.version}</version>
</dependency>
<!-- AspectJ Weaver for AOP -->
<dependency>
 <groupId>org.aspectj</groupId>
```



```
<artifactId>aspectjweaver</artifactId>
<version>1.9.22.1</version>
</dependency>
<!-- SLF4J API -->
<dependency>
 <groupId>org.slf4j</groupId>
 <artifactId>slf4j-api</artifactId>
 <version>${slf4j.version}</version>
</dependency>
<!-- Logback Classic -->
<dependency>
 <groupId>ch.qos.logback</groupId>
 <artifactId>logback-classic</artifactId>
 <version>${logback.version}</version>
</dependency>
<dependency>
 <groupId>javax.annotation</groupId>
 <artifactId>javax.annotation-api</artifactId>
 <version>1.3.2</version>
</dependency>
<dependency>
 <groupId>jakarta.servlet.jsp</groupId>
 <artifactId>jakarta.servlet.jsp-api</artifactId>
 <version>3.0.0</version>
 <scope>provided</scope>
</dependency>
<dependency>
 <groupId>javax.validation</groupId>
 <artifactId>validation-api</artifactId>
 <version>2.0.1.Final</version>
</dependency>
<dependency>
 <groupId>org.hibernate.validator</groupId>
 <artifactId>hibernate-validator</artifactId>
 <version>6.2.5.Final</version>
</dependency>
</dependencies>

<build>
 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-war-plugin</artifactId>
 <version>3.3.2</version>
 </plugin>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-compiler-plugin</artifactId>
 <configuration>
 <source>15</source>
 <target>15</target>
 </configuration>
 </plugin>
 </plugins>
</build>
</project>
```

- **web.xml:** Configures Spring DispatcherServlet and welcome file (index.jsp) for Servlet 6.0 (Jakarta EE 10).



```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
 version="6.0">

 <servlet>
 <servlet-name>dispatcher</servlet-name>
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
 <init-param>
 <param-name>contextClass</param-name>
 <param-
value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-
value>
 </init-param>
 <init-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>com.example.config.SpringConfig</param-value>
 </init-param>
 <load-on-startup>1</load-on-startup>
 </servlet>

 <servlet-mapping>
 <servlet-name>dispatcher</servlet-name>
 <url-pattern>/</url-pattern>
 </servlet-mapping>
</web-app>
```

- **SpringConfig.java:** Spring configuration with MVC, component scanning, HikariCP, and JSP view resolver.

```
• package com.example.config;

import com.example.aspect.LoggingAspect;
import com.zaxxer.hikari.HikariDataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.web.servlet.ViewResolver;
import
org.springframework.web.servlet.config.annotation.EnableWebMvc;
import
org.springframework.web.servlet.config.annotation.ResourceHandlerRegi
stry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import
```



```

org.springframework.web.servlet.view.InternalResourceViewResolver;

import javax.sql.DataSource;

@Configuration
@ComponentScan(basePackages = "com.example")
@EnableWebMvc
@EnableAspectJAutoProxy
public class SpringConfig implements WebMvcConfigurer {

 @Bean
 public DataSource dataSource() {
 HikariDataSource dataSource = new HikariDataSource();
 dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");

 dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/studentdb?useSSL=false&serverTimezone=UTC");
 dataSource.setUsername("root"); // Replace with your MySQL username
 dataSource.setPassword("Archer@12345"); // Replace with your MySQL password
 dataSource.setMaximumPoolSize(10);
 dataSource.setMinimumIdle(2);
 return dataSource;
 }

 @Bean
 public JdbcTemplate jdbcTemplate(DataSource dataSource) {
 return new JdbcTemplate(dataSource);
 }

 @Bean
 public ViewResolver viewResolver() {
 InternalResourceViewResolver resolver = new InternalResourceViewResolver();
 resolver.setPrefix("/WEB-INF/views/");
 resolver.setSuffix(".jsp");
 return resolver;
 }

 @Override
 public void addResourceHandlers(ResourceHandlerRegistry registry) {
 registry.addResourceHandler("/css/**").addResourceLocations("/css/");
 }
 @Bean
 public LoggingAspect loggingAspect() {
 return new LoggingAspect();
 }
}

```

- **Student.java:** Model class with student attributes (id, name, email) and Jakarta validation annotations.

```

package com.example.model;

import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;

```



```

public class Student {
 private int id;

 @NotBlank(message = "Name is required")
 @Size(max = 100, message = "Name must be less than 100
characters")
 private String name;

 @NotBlank(message = "Email is required")
 @Email(message = "Invalid email format")
 @Size(max = 255, message = "Email must be less than 255
characters")
 private String email;

 public Student() {}

 public Student(int id, String name, String email) {
 this.id = id;
 this.name = name;
 this.email = email;
 }

 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; }
}

```

- **StudentDAO.java:** Interface for CRUD operations on student records using Spring JDBC.

```

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();
}

```

- **StudentDAOImpl.java:** Implements StudentDAO with JdbcTemplate for database interactions.

```

package com.example.dao;

import com.example.annotation.Loggable;
import com.example.model.Student;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import java.sql.ResultSet;
import java.sql.SQLException;

```



```
import java.util.List;

@Repository
public class StudentDAOImpl implements StudentDAO {
 private final JdbcTemplate jdbcTemplate;

 @Autowired
 public StudentDAOImpl(JdbcTemplate jdbcTemplate) {
 this.jdbcTemplate = jdbcTemplate;
 }

 private final RowMapper<Student> studentRowMapper = new
 RowMapper<Student>() {
 @Override
 public Student mapRow(ResultSet rs, int rowNum) throws
 SQLException {
 return new Student(
 rs.getInt("id"),
 rs.getString("name"),
 rs.getString("email")
);
 }
 };
}

@Override
@Loggable
public void create(Student student) {
 String sql = "INSERT INTO student (name, email) VALUES (?, ?)";
 jdbcTemplate.update(sql, student.getName(),
 student.getEmail());
}

@Override
@Loggable
public Student read(int id) {
 String sql = "SELECT * FROM student WHERE id = ?";
 try {
 return jdbcTemplate.queryForObject(sql, new Object[]{id},
 studentRowMapper);
 } catch (Exception e) {
 return null;
 }
}

@Override
@Loggable
public void update(Student student) {
 String sql = "UPDATE student SET name = ?, email = ? WHERE id
 = ?";
 jdbcTemplate.update(sql, student.getName(),
 student.getEmail(), student.getId());
}

@Override
@Loggable
public void delete(int id) {
 String sql = "DELETE FROM student WHERE id = ?";
 jdbcTemplate.update(sql, id);
}
```



```
 @Override
 @Loggable
 public List<Student> getAll() {
 String sql = "SELECT * FROM student";
 return jdbcTemplate.query(sql, studentRowMapper);
 }
}
```

- **StudentController.java:** Handles HTTP requests for student CRUD and log viewing with Spring MVC.

```
package com.example.controller;
```

```
import com.example.annotation.Loggable;
import com.example.dao.StudentDAO;
import com.example.model.Student;
import com.example.util.LogViewer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.*;

import javax.validation.Valid;
import java.util.List;

@Controller
@RequestMapping("/students")
public class StudentController {
 private static final Logger logger = LoggerFactory.getLogger(StudentController.class);
 private final StudentDAO studentDAO;
 private final LogViewer logViewer;

 @Autowired
 public StudentController(StudentDAO studentDAO, LogViewer logViewer) {
 this.studentDAO = studentDAO;
 this.logViewer = logViewer;
 }

 @Loggable
 @GetMapping
 public String listStudents(Model model) {
 logger.debug("Listing students");
 List<Student> students = studentDAO.getAll();
 model.addAttribute("students", students);
 return "list";
 }
}
```



```
}

@Loggable
@GetMapping("/create")
public String showcreateForm(Model model) {
 model.addAttribute("student", new Student());
 return "create";
}

@Loggable
@PostMapping("/create")
public String createStudent(@Valid @ModelAttribute("student") Student student, BindingResult
result, Model model) {
 if (result.hasErrors()) {
 return "create";
 }
 try {
 studentDAO.create(student);
 return "redirect:/students";
 } catch (Exception e) {
 model.addAttribute("error", "Error creating student: " + e.getMessage());
 return "create";
 }
}

@Loggable
@GetMapping("/{id}")
public String viewStudent(@PathVariable("id") int id, Model model) {
 Student student = studentDAO.read(id);
 if (student == null) {
 model.addAttribute("error", "Student not found");
 }
 model.addAttribute("student", student);
 return "view";
}

@Loggable
@GetMapping("/edit/{id}")
public String showEditForm(@PathVariable("id") int id, Model model) {
 Student student = studentDAO.read(id);
 if (student == null) {
 model.addAttribute("error", "Student not found");
 return "edit";
 }
 model.addAttribute("student", student);
 return "edit";
}
```



```
@Loggable
@PostMapping("/edit/{id}")
public String updateStudent(@PathVariable("id") int id, @Valid @ModelAttribute("student")
Student student, BindingResult result, Model model) {
 if (result.hasErrors()) {
 return "edit";
 }
 student.setId(id);
 try {
 studentDAO.update(student);
 return "redirect:/students";
 } catch (Exception e) {
 model.addAttribute("error", "Error updating student: " + e.getMessage());
 return "edit";
 }
}

@Loggable
@GetMapping("/delete/{id}")
public String deleteStudent(@PathVariable("id") int id, Model model) {
 try {
 studentDAO.delete(id);
 return "redirect:/students";
 } catch (Exception e) {
 model.addAttribute("error", "Error deleting student: " + e.getMessage());
 return "list";
 }
}

@Loggable
@GetMapping("/logs")
public String viewLogs(Model model) {
 try {
 String logs = logViewer.readLogFile();
 model.addAttribute("logs", logs);
 } catch (Exception e) {
 model.addAttribute("error", "Error reading logs: " + e.getMessage());
 }
 return "logs";
}
```

- **LogViewer.java:** Reads logs from student-crud-web/logs/student-crud.log for display.

```
package com.example.util;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;
```



```

import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Collectors;

@Component
public class LogViewer {
 private static final Logger logger =
LoggerFactory.getLogger(LogViewer.class);
 private static final String LOG_FILE_PATH =
System.getProperty("user.dir") + "/log\\student-crud.log";

 public String readLogFile() {
 try {
 return Files.lines(Paths.get(LOG_FILE_PATH))
 .collect(Collectors.joining("\n"));
 } catch (Exception e) {
 logger.error("Error reading log file at {}: {}", LOG_FILE_PATH, e.getMessage());
 return "Unable to read logs: " + e.getMessage();
 }
 }
}

```

- **LoggingAspect.java:** AOP aspect for logging method entry, exit, and exceptions.

```

package com.example.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Arrays;

@Aspect
public class LoggingAspect {
 private static final Logger logger =
LoggerFactory.getLogger(LoggingAspect.class);

 @Around("@annotation(com.example.annotation.Loggable)")
 public Object logMethod(ProceedingJoinPoint joinPoint) throws
Throwable {
 String methodName = joinPoint.getSignature().getName();
 String className =
joinPoint.getTarget().getClass().getSimpleName();
 Object[] args = joinPoint.getArgs();

 logger.trace("Entering {}.{} with arguments: {}", className,
methodName, Arrays.toString(args));

 try {
 Object result = joinPoint.proceed();
 logger.info("Exiting {}.{} with result: {}", className,
methodName, result);
 return result;
 } catch (Throwable e) {
 logger.error("Exception in {}.{}: {}", className,
methodName, e.getMessage(), e);
 throw e;
 }
 }
}

```



```

 }
 }
}
```

- **Loggable.java:** Custom annotation to mark methods for AOP logging.

```

package com.example.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Loggable {
}
```

- **DatabaseInitializer.java:** Initializes MySQL student table on application startup.

```

package com.example.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;

@Component
public class DatabaseInitializer {
 private final JdbcTemplate jdbcTemplate;

 @Autowired
 public DatabaseInitializer(JdbcTemplate jdbcTemplate) {
 this.jdbcTemplate = jdbcTemplate;
 }

 @PostConstruct
 public void initialize() {
 String sql = """
 CREATE TABLE IF NOT EXISTS student (
 id INT PRIMARY KEY AUTO_INCREMENT,
 name VARCHAR(100) NOT NULL,
 email VARCHAR(255) NOT NULL UNIQUE
)
 """;
 jdbcTemplate.execute(sql);
 }
}
```

- **index.jsp:** Welcome page with links to student management and logs.

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
```



```

<head>
 <title>Student Management System</title>
 <link rel="stylesheet"
 href="\${pageContext.request.contextPath}/css/style.css">
</head>
<body>
<div class="container">
 <h1>Welcome to Student Management System</h1>
 <nav>
 Manage
 Students
 View Logs
 </nav>
</div>
</body>
</html>

```

- **list.jsp:** Displays a table of students with view, edit, and delete options.

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="jakarta.tags.core" prefix="c" %>
<html>
<head>
 <title>Student List</title>
 <link rel="stylesheet"
 href="\${pageContext.request.contextPath}/css/style.css">
</head>
<body>
<div class="container">
 <h1>Student List</h1>
 <nav>
 Create New
 Student
 View Logs
 Home
 </nav>
 <c:if test="\${not empty error}">
 <p class="error">\${error}</p>
 </c:if>
 <table>
 <tr>
 <th>ID</th>
 <th>Name</th>
 <th>Email</th>
 <th>Actions</th>
 </tr>
 <c:forEach var="student" items="\${students}">
 <tr>
 <td>\${student.id}</td>
 <td>\${student.name}</td>
 <td>\${student.email}</td>
 <td>
 View

 Edit
 </td>
 </tr>
 </c:forEach>
 </table>
</div>

```



```
 href="${pageContext.request.contextPath}/students/delete/${student.id}"
 <td>
 </td>
 </tr>
 </c:forEach>
</table>
</div>
</body>
</html>
```

- **create.jsp:** Form for creating a new student with validation.

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
 <title>Create Student</title>
 <link rel="stylesheet" href="#">href="${pageContext.request.contextPath}/css/style.css"href="${pageContext.request.contextPath}/students"href="${pageContext.request.contextPath}/students/logs"href="${pageContext.request.contextPath}/"test="${not empty error}"error</p>
 </c:if>
 <form:form modelAttribute="student" method="post"
action="#">action="${pageContext.request.contextPath}/students/create"
```



- **edit.jsp:** Form for updating an existing student's details.

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
 <title>Edit Student</title>
 <link rel="stylesheet"
 href="${pageContext.request.contextPath}/css/style.css">
</head>
<body>
<div class="container">
 <h1>Edit Student</h1>
 <nav>
 Back to List
 View Logs
 Home
 </nav>
 <c:if test="${not empty error}">
 <p class="error">${error}</p>
 </c:if>
 <form:form modelAttribute="student" method="post"
 action="${pageContext.request.contextPath}/students/edit/${student.id}">
 <div>
 <label for="name">Name:</label>
 <form:input path="name" id="name"/>
 <form:errors path="name" cssClass="error-field"/>
 </div>
 <div>
 <label for="email">Email:</label>
 <form:input path="email" id="email" type="email"/>
 <form:errors path="email" cssClass="error-field"/>
 </div>
 <button type="submit">Update</button>
 </form:form>
</div>
</body>
</html>
```

- **view.jsp:** Displays details of a single student.

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
 <title>View Student</title>
 <link rel="stylesheet"
 href="${pageContext.request.contextPath}/css/style.css">
</head>
<body>
<div class="container">
 <h1>Student Details</h1>
 <nav>
 Back to
```



```

List
 ${pageContext.request.contextPath}/students/logs ${pageContext.request.contextPath}/ ${not empty error} ${error}</p>
 </c:if>
 <c:if test="#"> ${not empty student} ${student.id}</td>
 </tr>
 <tr>
 <th>Name</th>
 <td> ${student.name}</td>
 </tr>
 <tr>
 <th>Email</th>
 <td> ${student.email}</td>
 </tr>
 </table>
 </c:if>
 </div>
</body>
</html>

```

- **logs.jsp:** Shows application logs from LogViewer with error handling.
- ```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="jakarta.tags.core" prefix="c" %>
<html>
    <head>
        <title>View Logs</title>
        <link href="#"> ${pageContext.request.contextPath}/css/style.css ${pageContext.request.contextPath}/students ${pageContext.request.contextPath}/ ${logs.startsWith('Unable to')} ${logs}" /></p>
                </c:when>
                <c:otherwise>
                    <pre><c:out value=" ${logs}" /></pre>
                </c:otherwise>
            </c:choose>
        </div>
    </body>
</html>

```



- **style.css:** CSS styles for consistent UI across JSP pages.

```
body {  
    font-family: Arial, sans-serif;  
    margin: 0;  
    padding: 20px;  
    background-color: #f4f4f4;  
}  
h1, h2 {  
    color: #333;  
}  
.container {  
    max-width: 800px;  
    margin: 0 auto;  
    background: #fff;  
    padding: 20px;  
    border-radius: 8px;  
    box-shadow: 0 0 10px rgba(0,0,0,0.1);  
}  
table {  
    width: 100%;  
    border-collapse: collapse;  
    margin-bottom: 20px;  
}  
th, td {  
    padding: 10px;  
    border: 1px solid #ddd;  
    text-align: left;  
}  
th {  
    background-color: #4CAF50;  
    color: white;  
}  
tr:nth-child(even) {  
    background-color: #f2f2f2;  
}  
a {  
    color: #4CAF50;  
    text-decoration: none;  
    margin-right: 10px;  
}  
a:hover {  
    text-decoration: underline;  
}  
form {  
    display: flex;  
    flex-direction: column;  
    gap: 10px;  
}  
input[type="text"], input[type="email"] {  
    padding: 8px;  
    border: 1px solid #ddd;  
    border-radius: 4px;  
}  
button {  
    padding: 10px;  
    background-color: #4CAF50;  
    color: white;  
    border: none;  
    border-radius: 4px;  
    cursor: pointer;  
}
```



```
button:hover {  
    background-color: #45a049;  
}  
.error {  
    color: red;  
    font-weight: bold;  
}  
pre {  
    background: #f8f8f8;  
    padding: 10px;  
    border: 1px solid #ddd;  
    border-radius: 4px;  
    max-height: 400px;  
    overflow-y: auto;  
}  
nav {  
    margin-bottom: 20px;  
}  
nav a {  
    margin-right: 20px;  
    font-weight: bold;  
}  
.error-field {  
    color: red;  
    font-size: 0.9em;  
}
```

- **logback.xml:** Configures Logback to write logs to student-crud-web/logs/student-crud.log.

```
<configuration>  
    <property name="LOG_DIR" value="${user.dir}/log" />  
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">  
        <encoder>  
            <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level [%logger{36}] - %msg%n</pattern>  
        </encoder>  
    </appender>  
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">  
        <file>${LOG_DIR}/student-crud.log</file>  
        <append>true</append>  
        <encoder>  
            <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level [%logger{36}] - %msg%n</pattern>  
        </encoder>  
    </appender>  
    <logger name="com.example" level="DEBUG" additivity="false">  
        <appender-ref ref="CONSOLE"/>  
        <appender-ref ref="FILE"/>  
    </logger>  
    <logger name="org.springframework" level="DEBUG" additivity="false">  
        <appender-ref ref="CONSOLE"/>  
        <appender-ref ref="FILE"/>  
    </logger>  
    <root level="INFO">
```



```

<appender-ref ref="CONSOLE"/>
<appender-ref ref="FILE"/>
</root>
</configuration>

```

Welcome to Student Management System

[Manage Students](#) [View Logs](#)

Student List

[Create New Student](#) [View Logs](#) [Home](#)

ID	Name	Email	Actions
1	John More	john.doe@example.com	View Edit Delete
3	Mark Devid	md@live.com	View Edit Delete
4	Mike Bostan	mb@gmail.com	View Edit Delete
5	lisa pawar	pawarlisa@gmail.com	View Edit Delete

Student Details

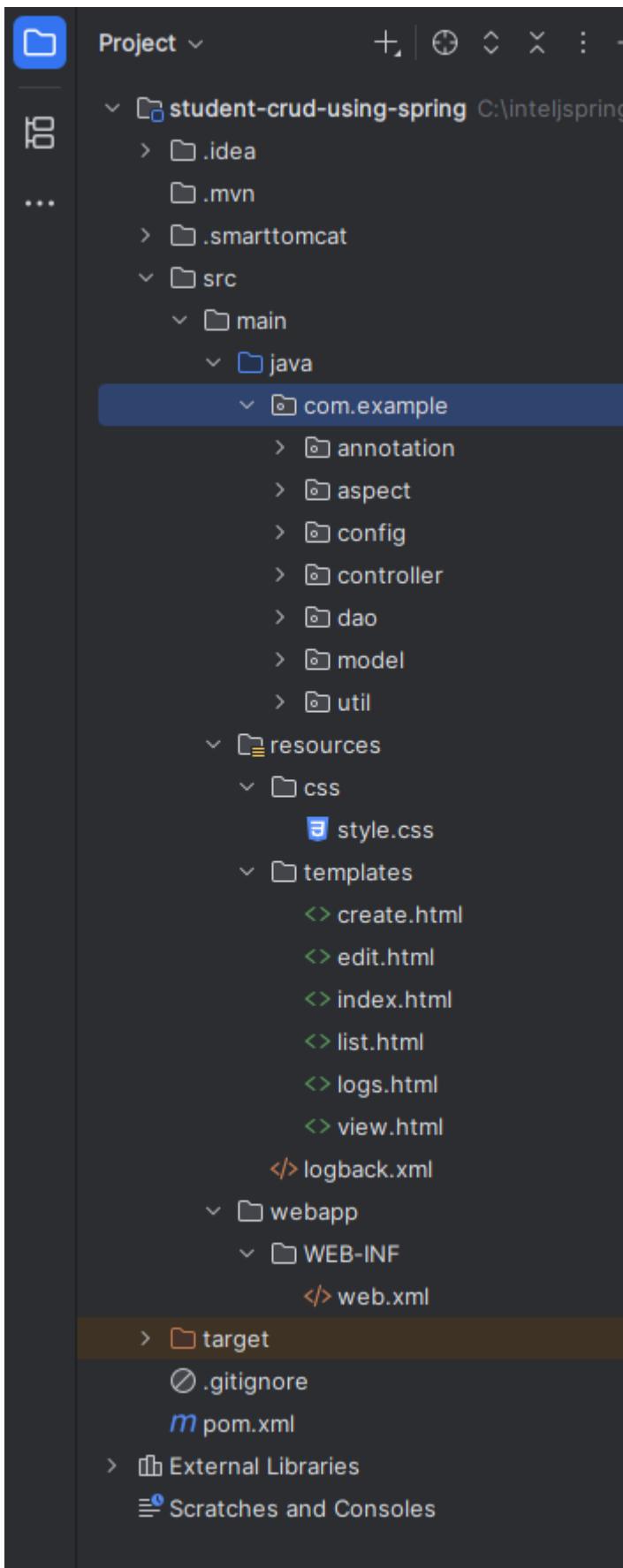
[Back to List](#) [View Logs](#) [Home](#)

ID	1
Name	John More
Email	john.doe@example.com

Access URLs:

- **Home:** <http://localhost:8080/student-crud-web/>
- **List Students:** <http://localhost:8080/student-crud-web/students>
- **Create Student:** <http://localhost:8080/student-crud-web/students/create>
- **View Student:** <http://localhost:8080/student-crud-web/students/{id}>
- **Edit Student:** <http://localhost:8080/student-crud-web/students/edit/{id}>
- **Delete Student:** <http://localhost:8080/student-crud-web/students/delete/{id}>
- **View Logs:** <http://localhost:8080/student-crud-web/students/logs>



Using Thymeleaf for UI: Same above example with Thymeleaf used for UI

- **pom.xml**

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.example</groupId>
  <artifactId>student-crud-using-spring</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>student-crud-using-spring Maven Webapp</name>
  <url>http://maven.apache.org</url>

  <properties>
    <java.version>21</java.version>
    <spring.version>6.1.14</spring.version>
    <thymeleaf.version>3.1.2.RELEASE</thymeleaf.version>
    <slf4j.version>2.0.16</slf4j.version>
    <logback.version>1.5.12</logback.version>
  </properties>

  <dependencies>
    <!-- Spring Core -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <!-- Spring MVC -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <!-- Spring JDBC -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>${spring.version}</version>
    </dependency>

    <!-- Servlet API (for web application) -->
    <dependency>
      <groupId>jakarta.servlet</groupId>
      <artifactId>jakarta.servlet-api</artifactId>
      <version>6.0.0</version>
      <scope>provided</scope>
    </dependency>
    <!-- MySQL Connector -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.33</version>
    </dependency>
    <!-- HikariCP -->
    <dependency>
```



```
<groupId>com.zaxxer</groupId>
<artifactId>HikariCP</artifactId>
<version>5.1.0</version>
</dependency>
<!-- Thymeleaf -->
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf</artifactId>
    <version>${thymeleaf.version}</version>
</dependency>
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf-spring6</artifactId>
    <version>${thymeleaf.version}</version>
</dependency>
<!-- Spring AOP -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>${spring.version}</version>
</dependency>
<!-- AspectJ Weaver for AOP -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.22.1</version>
</dependency>
<!-- SLF4J API -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>${slf4j.version}</version>
</dependency>
<!-- Logback Classic -->
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>${logback.version}</version>
</dependency>
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>2.0.1.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.2.5.Final</version>
</dependency>
<dependency>
    <groupId>jakarta.validation</groupId>
    <artifactId>jakarta.validation-api</artifactId>
    <version>3.0.2</version>
</dependency>

</dependencies>
```



```
<build>
  <finalName>student-crud-using-spring</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>15</source>
        <target>15</target>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

- **web.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
  version="5.0">

  <display-name>student-crud-using-spring</display-name>

  <!-- Spring DispatcherServlet -->
  <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
    <init-param>
      <param-name>contextClass</param-name>
      <param-
value>org.springframework.web.context.support.AnnotationConfigWebAppl
icationContext</param-value>
    </init-param>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-
value>com.example.studentcrudusingspring.config.WebConfig</param-
value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

  <!-- Spring Context Listener -->
  <listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-
class>
  </listener>

  <context-param>
    <param-name>contextClass</param-name>
```



```

<param-
value>org.springframework.web.context.support.AnnotationConfigWebAppl
icationContext</param-value>
</context-param>
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>com.example.config.SpringConfig</param-value>
</context-param>
</web-app>

```

- **SpringConfig.java**

```

• package com.example.config;

import com.example.aspect.LoggingAspect;
import com.zaxxer.hikari.HikariDataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;
import org.springframework.jdbc.core.JdbcTemplate;
import
org.springframework.validation.beanvalidation.LocalValidatorFactoryBe
an;
import
org.springframework.web.servlet.config.annotation.EnableWebMvc;
import
org.springframework.web.servlet.config.annotation.ResourceHandlerRegi
stry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.thymeleaf.spring6.SpringTemplateEngine;
import org.thymeleaf.spring6.view.ThymeleafViewResolver;
import
org.thymeleaf.spring6.templateresolver.SpringResourceTemplateResolver
;

import javax.sql.DataSource;

@Configuration
@ComponentScan(basePackages = "com.example")
@EnableWebMvc
@EnableAspectJAutoProxy
public class SpringConfig implements WebMvcConfigurer {

    @Bean
    public DataSource dataSource() {
        HikariDataSource dataSource = new HikariDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");

        dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/studentdb?useSSL=f
alse&serverTimezone=UTC");
        dataSource.setUsername("root"); // Replace with your MySQL
username
        dataSource.setPassword("Archer@12345"); // Replace with your
MySQL password
        dataSource.setMaximumPoolSize(10);
        dataSource.setMinimumIdle(2);
        return dataSource;
    }
}

```



```
@Bean
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
    return new JdbcTemplate(dataSource);
}

@Bean
public SpringResourceTemplateResolver templateResolver() {
    SpringResourceTemplateResolver resolver = new
SpringResourceTemplateResolver();
    resolver.setPrefix("classpath:/templates/");
    resolver.setSuffix(".html");
    resolver.setTemplateMode("HTML");
    return resolver;
}

@Bean
public SpringTemplateEngine templateEngine() {
    SpringTemplateEngine engine = new SpringTemplateEngine();
    engine.setTemplateResolver(templateResolver());
    return engine;
}

@Bean
public ThymeleafViewResolver viewResolver() {
    ThymeleafViewResolver resolver = new ThymeleafViewResolver();
    resolver.setTemplateEngine(templateEngine());
    resolver.setCharacterEncoding("UTF-8");
    return resolver;
}

@Override
public void addResourceHandlers(ResourceHandlerRegistry registry)
{
    registry.addResourceHandler("/css/**").addResourceLocations("classpath:/css/");
}
@Bean
public LoggingAspect loggingAspect() {
    return new LoggingAspect();
}
```

- **Student.java**

- package com.example.model;

```
import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;

public class Student {
    private int id;

    @NotBlank(message = "Name is required")
    @Size(max = 100, message = "Name must be less than 100
characters")
    private String name;

    @NotBlank(message = "Email is required")
```



```

    @Email(message = "Invalid email format")
    @Size(max = 255, message = "Email must be less than 255
characters")
    private String email;

    public Student() {}

    public Student(int id, String name, String email) {
        this.id = id;
        this.name = name;
        this.email = email;
    }

    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; }
}

```

- **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();
}

```

- **StudentDAOImpl.java**

```

• package com.example.dao;

import com.example.annotation.Loggable;
import com.example.model.Student;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

@Repository
public class StudentDAOImpl implements StudentDAO {
    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public StudentDAOImpl(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    private final RowMapper<Student> studentRowMapper = new
    RowMapper<Student>() {

```



```

        @Override
        public Student mapRow(ResultSet rs, int rowNum) throws
SQLException {
    return new Student(
        rs.getInt("id"),
        rs.getString("name"),
        rs.getString("email")
    );
}
};

@Override
@Loggable
public void create(Student student) {
    String sql = "INSERT INTO student (name, email) VALUES (?, ?)";
    jdbcTemplate.update(sql, student.getName(),
student.getEmail());
}

@Override
@Loggable
public Student read(int id) {
    String sql = "SELECT * FROM student WHERE id = ?";
    try {
        return jdbcTemplate.queryForObject(sql, new Object[]{id},
studentRowMapper);
    } catch (Exception e) {
        return null;
    }
}

@Override
@Loggable
public void update(Student student) {
    String sql = "UPDATE student SET name = ?, email = ? WHERE id
= ?";
    jdbcTemplate.update(sql, student.getName(),
student.getEmail(), student.getId());
}

@Override
@Loggable
public void delete(int id) {
    String sql = "DELETE FROM student WHERE id = ?";
    jdbcTemplate.update(sql, id);
}

@Override
@Loggable
public List<Student> getAll() {
    String sql = "SELECT * FROM student";
    return jdbcTemplate.query(sql, studentRowMapper);
}
}

```

- **StudentController.java**

```

package com.example.controller;

import com.example.annotation.Loggable;
```



```
import com.example.dao.StudentDAO;
import com.example.model.Student;
import com.example.util.LogViewer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.*;

import javax.validation.Valid;
import java.util.ArrayList;
import java.util.List;

@Controller
public class StudentController {
    private static final Logger logger =
LoggerFactory.getLogger(StudentController.class);
    private final StudentDAO studentDAO;
    private final LogViewer logViewer;

    @Autowired
    public StudentController(StudentDAO studentDAO, LogViewer
logViewer) {
        this.studentDAO = studentDAO;
        this.logViewer = logViewer;
    }

    @GetMapping("/")
    public String test(Model model) {
        logger.debug("index endpoint called");
        return "index";
    }

    @Loggable
    @GetMapping("/students")
    public String listStudents(Model model) {
        logger.debug("Listing students");
        List<Student> students = studentDAO.getAll();
        model.addAttribute("students", students);
        return "list";
    }

    @Loggable
    @GetMapping("/students/create")
    public String showCreateForm(Model model) {
        model.addAttribute("student", new Student());
        return "create";
    }

    @Loggable
    @PostMapping("/students/create")
    public String createStudent(@Valid @ModelAttribute("student")
Student student, BindingResult result, Model model) {
        if (result.hasErrors()) {
            return "create";
        }
        try {
            studentDAO.create(student);
            return "redirect:/students";
        }
    }
}
```



```
        } catch (Exception e) {
            model.addAttribute("error", "Error creating student: " +
e.getMessage());
            return "create";
        }
    }

@Loggable
@GetMapping("/students/{id}")
public String viewStudent(@PathVariable("id") int id, Model
model) {
    Student student = studentDAO.read(id);
    if (student == null) {
        model.addAttribute("error", "Student not found");
    }
    model.addAttribute("student", student);
    return "view";
}

@Loggable
@GetMapping("/students/edit/{id}")
public String showEditForm(@PathVariable("id") int id, Model
model) {
    Student student = studentDAO.read(id);
    if (student == null) {
        model.addAttribute("error", "Student not found");
        return "edit";
    }
    model.addAttribute("student", student);
    return "edit";
}

@Loggable
@PostMapping("/students/edit/{id}")
public String updateStudent(@PathVariable("id") int id, @Valid
@ModelAttribute("student") Student student, BindingResult result,
Model model) {
    if (result.hasErrors()) {
        return "edit";
    }
    student.setId(id);
    try {
        studentDAO.update(student);
        return "redirect:/students";
    } catch (Exception e) {
        model.addAttribute("error", "Error updating student: " +
e.getMessage());
        return "edit";
    }
}

@Loggable
@GetMapping("/students/delete/{id}")
public String deleteStudent(@PathVariable("id") int id, Model
model) {
    try {
        studentDAO.delete(id);
        return "redirect:/students";
    } catch (Exception e) {
        model.addAttribute("error", "Error deleting student: " +
e.getMessage());
    }
}
```



```

        return "list";
    }
}

@Loggable
@GetMapping("/students/logs")
public String viewLogs(Model model) {
    try {
        String logs = logViewer.readLogFile();
        model.addAttribute("logs", logs);
    } catch (Exception e) {
        model.addAttribute("error", "Error reading logs: " +
e.getMessage());
    }
    return "logs";
}
}

```

- **LogViewer.java**

```

● package com.example.util;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.stream.Collectors;

@Component
public class LogViewer {
    private static final Logger logger =
LoggerFactory.getLogger(LogViewer.class);
    private static final String LOG_FILE_PATH =
System.getProperty("user.dir") + "/logs/student-crud.log";

    public String readLogFile() {
        try {
            Path logDir = Paths.get(System.getProperty("user.dir") +
"/logs");
            logger.debug("Attempting to create log directory: {}", logDir);
            Files.createDirectories(logDir);
            Path logFile = Paths.get(LOG_FILE_PATH);
            logger.debug("Reading log file from: {}", logFile);
            if (Files.exists(logFile)) {
                return
Files.lines(logFile).collect(Collectors.joining("\n"));
            } else {
                logger.warn("Log file does not exist: {}", logFile);
                return "Log file not found at: " + LOG_FILE_PATH;
            }
        } catch (Exception e) {
            logger.error("Error reading log file at {}: {}", LOG_FILE_PATH, e.getMessage(), e);
            return "Unable to read logs: " + LOG_FILE_PATH + " - " +
e.getMessage();
        }
    }
}

```



```
    }
```

- **LoggingAspect.java**

```
package com.example.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Arrays;

@Aspect
public class LoggingAspect {
    private static final Logger logger =
    LoggerFactory.getLogger(LoggingAspect.class);

    @Around("@annotation(com.example.annotation.Loggable)")
    public Object logMethod(ProceedingJoinPoint joinPoint) throws
Throwable {
        String methodName = joinPoint.getSignature().getName();
        String className =
joinPoint.getTarget().getClass().getSimpleName();
        Object[] args = joinPoint.getArgs();

        logger.trace("Entering {}.{ } with arguments: {}", className,
methodName, Arrays.toString(args));

        try {
            Object result = joinPoint.proceed();
            logger.info("Exiting {}.{ } with result: {}", className,
methodName, result);
            return result;
        } catch (Throwable e) {
            logger.error("Exception in {}.{ }: {}", className,
methodName, e.getMessage(), e);
            throw e;
        }
    }
}
```

- **Loggable.java**

```
package com.example.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Loggable {
}
```

- **DatabaseInitializer.java**

```
package com.example.config;
```



```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;

@Component
public class DatabaseInitializer {
    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public DatabaseInitializer(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @PostConstruct
    public void initialize() {
        String sql = """
            CREATE TABLE IF NOT EXISTS student (
                id INT PRIMARY KEY AUTO_INCREMENT,
                name VARCHAR(100) NOT NULL,
                email VARCHAR(255) NOT NULL UNIQUE
            )
            """;
        jdbcTemplate.execute(sql);
    }
}

```

- **index.html**

```

• <!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Welcome</title>
    <link th:href="@{/css/style.css}" rel="stylesheet"/>
</head>
<body>
<div class="container">
    <h1>Welcome to Student Management System</h1>
    <nav>
        <a th:href="@{/students}">View Students</a>
        <a th:href="@{/students/logs}">View Logs</a>
    </nav>
</div>
</body>
</html>

```

- **list.html**

```

• <!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Student List</title>
    <link th:href="@{/css/style.css}" rel="stylesheet"/>
</head>
<body>
<div class="container">
    <h1>Student List</h1>
    <nav>
        <a th:href="@{/students/create}">Create New Student</a>
    </nav>

```



```

        <a th:href="@{/students/logs}">View Logs</a>
        <a th:href="@{ / }">Home</a>
    </nav>
    <div th:if="${error}" class="error" th:text="${error}"></div>
    <table>
        <tr>
            <th>ID</th>
            <th>Name</th>
            <th>Email</th>
            <th>Actions</th>
        </tr>
        <tr th:each="student : ${students}">
            <td th:text="${student.id}"></td>
            <td th:text="${student.name}"></td>
            <td th:text="${student.email}"></td>
            <td>
                <a th:href="@{/students/{id} (id=${student.id}) }">View</a>
                <a th:href="@{/students/edit/{id} (id=${student.id}) }">Edit</a>
                <a th:href="@{/students/delete/{id} (id=${student.id}) }" onclick="return confirm('Are you sure?')">Delete</a>
            </td>
        </tr>
    </table>
</div>
</body>
</html>

```

- **create.html**

```

• <!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Create Student</title>
    <link th:href="@{/css/style.css}" rel="stylesheet"/>
</head>
<body>
<div class="container">
    <h1>Create Student</h1>
    <nav>
        <a th:href="@{/students}">Back to List</a>
        <a th:href="@{/students/logs}">View Logs</a>
        <a th:href="@{ / }">Home</a>
    </nav>
    <form th:action="@{/students/create}" th:object="${student}"
method="post">
        <div>
            <label for="name">Name:</label>
            <input type="text" id="name" th:field="*{name}" />
            <span th:errors="*{name}" class="error"></span>
        </div>
        <div>
            <label for="email">Email:</label>
            <input type="text" id="email" th:field="*{email}" />
            <span th:errors="*{email}" class="error"></span>
        </div>
        <button type="submit">Create</button>
    </form>

```



```
</div>
</body>
</html>
```

- **edit.html**

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Edit Student</title>
    <link th:href="@{/css/style.css}" rel="stylesheet"/>
</head>
<body>
<div class="container">
    <h1>Edit Student</h1>
    <nav>
        <a th:href="@{/students}">Back to List</a>
        <a th:href="@{/students/logs}">View Logs</a>
        <a th:href="@{ / }">Home</a>
    </nav>
    <form th:action="@{/students/edit/{id} (id=${student.id})}" th:object="${student}" method="post">
        <input type="hidden" th:field="*{id}" />
        <div>
            <label for="name">Name:</label>
            <input type="text" id="name" th:field="*{name}" />
            <span th:errors="*{name}" class="error"></span>
        </div>
        <div>
            <label for="email">Email:</label>
            <input type="text" id="email" th:field="*{email}" />
            <span th:errors="*{email}" class="error"></span>
        </div>
        <button type="submit">Update</button>
    </form>
</div>
</body>
</html>
```

- **view.html**

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>View Student</title>
    <link th:href="@{/css/style.css}" rel="stylesheet"/>
</head>
<body>
<div class="container">
    <h1>Student Details</h1>
    <nav>
        <a th:href="@{/students}">Back to List</a>
        <a th:href="@{/students/logs}">View Logs</a>
        <a th:href="@{ / }">Home</a>
    </nav>
    <div>
        <p><strong>ID:</strong> <span th:text="${student.id}"></span></p>
        <p><strong>Name:</strong> <span th:text="${student.name}"></span></p>
    </div>
</div>
</body>
</html>
```



```
<p><strong>Email:</strong> <span  
th:text="${student.email}"></span></p>  
</div>  
</div>  
</body>  
</html>
```

- **logs.html**

```
<!DOCTYPE html>  
<html lang="en" xmlns:th="http://www.thymeleaf.org">  
<head>  
    <meta charset="UTF-8">  
    <title>View Logs</title>  
    <link th:href="@{/css/style.css}" rel="stylesheet"/>  
</head>  
<body>  
<div class="container">  
    <h1>Application Logs</h1>  
    <nav>  
        <a th:href="@{/students}">Back to List</a>  
        <a th:href="@{ / }">Home</a>  
    </nav>  
    <div th:if="${logs.startsWith('Unable to')}">  
        <p class="error" th:text="${logs}"></p>  
    </div>  
    <div th:unless="${logs.startsWith('Unable to')}">  
        <pre th:text="${logs}"></pre>  
    </div>  
</div>  
</body>  
</html>
```

- **style.css**

```
body {  
    font-family: Arial, sans-serif;  
    margin: 0;  
    padding: 20px;  
    background-color: #f4f4f4;  
}  
h1, h2 {  
    color: #333;  
}  
.container {  
    max-width: 800px;  
    margin: 0 auto;  
    background: #fff;  
    padding: 20px;  
    border-radius: 8px;  
    box-shadow: 0 0 10px rgba(0,0,0,0.1);  
}  
table {  
    width: 100%;  
    border-collapse: collapse;  
    margin-bottom: 20px;  
}  
th, td {  
    padding: 10px;  
    border: 1px solid #ddd;  
    text-align: left;  
}
```



```
th {  
    background-color: #4CAF50;  
    color: white;  
}  
tr:nth-child(even) {  
    background-color: #f2f2f2;  
}  
a {  
    color: #4CAF50;  
    text-decoration: none;  
    margin-right: 10px;  
}  
a:hover {  
    text-decoration: underline;  
}  
form {  
    display: flex;  
    flex-direction: column;  
    gap: 10px;  
}  
input[type="text"], input[type="email"] {  
    padding: 8px;  
    border: 1px solid #ddd;  
    border-radius: 4px;  
}  
button {  
    padding: 10px;  
    background-color: #4CAF50;  
    color: white;  
    border: none;  
    border-radius: 4px;  
    cursor: pointer;  
}  
button:hover {  
    background-color: #45a049;  
}  
.error {  
    color: red;  
    font-weight: bold;  
}  
pre {  
    background: #f8f8f8;  
    padding: 10px;  
    border: 1px solid #ddd;  
    border-radius: 4px;  
    max-height: 400px;  
    overflow-y: auto;  
}  
nav {  
    margin-bottom: 20px;  
}  
nav a {  
    margin-right: 20px;  
    font-weight: bold;  
}  
.error-field {  
    color: red;  
    font-size: 0.9em;  
}
```



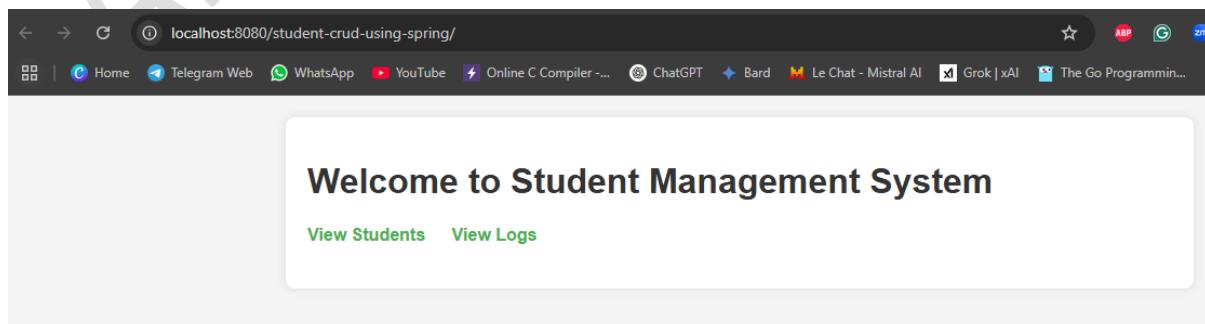
- **logback.xml**

```

• <configuration>
    <property name="LOG_DIR" value="${user.dir}/logs" />
    <appender name="CONSOLE"
    class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level [%logger{36}] -
            %msg%n</pattern>
        </encoder>
    </appender>
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>${LOG_DIR}/student-crud.log</file>
        <append>true</append>
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level [%logger{36}] -
            %msg%n</pattern>
        </encoder>
    </appender>
    <logger name="com.example" level="DEBUG" additivity="false">
        <appender-ref ref="CONSOLE"/>
        <appender-ref ref="FILE"/>
    </logger>
    <logger name="org.springframework" level="DEBUG"
    additivity="false">
        <appender-ref ref="CONSOLE"/>
        <appender-ref ref="FILE"/>
    </logger>
    <root level="INFO">
        <appender-ref ref="CONSOLE"/>
        <appender-ref ref="FILE"/>
    </root>
</configuration>
```

Accessible URLs

- <http://localhost:8080/student-crud-web/> - Displays the welcome page (index.html).
- <http://localhost:8080/student-crud-web/students> - Lists all students (list.html).
- <http://localhost:8080/student-crud-web/students/test> - Test endpoint, displays an empty student list (list.html).
- <http://localhost:8080/student-crud-web/students/create> - Shows the form to create a new student (create.html).
- <http://localhost:8080/student-crud-web/students/{id}> - Views details of a student by ID (view.html).
- <http://localhost:8080/student-crud-web/students/edit/{id}> - Shows the form to edit a student by ID (edit.html).
- <http://localhost:8080/student-crud-web/students/delete/{id}> - Deletes a student by ID, redirects to /students.
- <http://localhost:8080/student-crud-web/students/logs> - Displays application logs (logs.html).



localhost:8080/student-crud-using-spring/students

Student List

Create New Student View Logs Home

ID	Name	Email	Actions
1	John More	john.doe@example.com	View Edit Delete
3	Mark Devid	md@live.com	View Edit Delete
4	Mike Bostan	mb@gmail.com	View Edit Delete

localhost:8080/student-crud-using-spring/students/logs

Application Logs

Back to List Home

```
2025-06-25 10:20:05 INFO [o.s.web.context.ContextLoader] - Root WebApplicationContext: initialization start
2025-06-25 10:20:05 DEBUG [o.s.w.c.s.AnnotationConfigWebApplicationContext] - Refreshing Root WebApplication
2025-06-25 10:20:05 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:06 DEBUG [o.s.c.a.ClassPathBeanDefinitionScanner] - Identified candidate component class: f
2025-06-25 10:20:06 DEBUG [o.s.c.a.ClassPathBeanDefinitionScanner] - Identified candidate component class: f
2025-06-25 10:20:06 DEBUG [o.s.c.a.ClassPathBeanDefinitionScanner] - Identified candidate component class: f
2025-06-25 10:20:06 DEBUG [o.s.c.a.ClassPathBeanDefinitionScanner] - Identified candidate component class: f
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Unable to locate ThemeSource with name 'th
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:07 DEBUG [o.s.a.a.a.ReflectiveAspectJAdvisorFactory] - Found AspectJ method: public java.la
2025-06-25 10:20:07 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:07 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:07 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:07 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:07 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bean
2025-06-25 10:20:07 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Autowiring by type from bean name 'jdbcTe
2025-06-25 10:20:07 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Autowiring by type from bean name 'databa
2025-06-25 10:20:07 DEBUG [o.s.j.jdbc.core.JdbcTemplate] - Executing SQL statement [CREATE TABLE IF NOT EXISTS
id INT PRIMARY KEY AUTO_INCREMENT,
name VARCHAR(100) NOT NULL,
email VARCHAR(255) NOT NULL UNIQUE
]
]
```

localhost:8080/student-crud-using-spring/students/1

Student Details

Back to List View Logs Home

ID: 1

Name: John More

Email: john.doe@example.com

localhost:8080/student-crud-using-spring/students/edit/1

Edit Student

Back to List View Logs Home

Name:

Email:

[Update](#)



Row Mapping:

In Spring JDBC, Row Mapping refers to the process of mapping rows from a database query result (typically a ResultSet) to Java objects. This is a key feature of Spring's JDBC framework, which simplifies database operations by abstracting low-level JDBC details.

Key Points about Row Mapping in Spring JDBC:

1. Purpose: Row mapping converts each row of a ResultSet into a Java object, such as a POJO (Plain Old Java Object), making it easier to work with query results in an object-oriented way.
2. RowMapper Interface:
 - o Spring provides the RowMapper interface to define how a single row of the ResultSet is mapped to a Java object.
 - o You implement the RowMapper interface by overriding its mapRow(ResultSet rs, int rowNum) method.
 - o Example:

```
import org.springframework.jdbc.core.RowMapper;
import java.sql.ResultSet;
import java.sql.SQLException;

public class UserRowMapper implements RowMapper<User> {
    @Override
    public User mapRow(ResultSet rs, int rowNum) throws SQLException {
        User user = new User();
        user.setId(rs.getInt("id"));
        user.setName(rs.getString("name"));
        user.setEmail(rs.getString("email"));
        return user;
    }
}
```

Here, each row of the ResultSet is mapped to a User object.

3. Usage in JdbcTemplate:
 - o The RowMapper is typically used with JdbcTemplate methods like query() or queryForObject() to execute SQL queries and map results.
 - o Example:

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
List<User> users = jdbcTemplate.query("SELECT * FROM users", new UserRowMapper());
```

This retrieves all rows from the users table and maps each row to a User object.

4. Benefits:
 - o Simplifies data access by eliminating boilerplate code for iterating over a ResultSet.
 - o Allows developers to focus on object-oriented logic rather than low-level JDBC operations.
 - o Reusable RowMapper implementations for consistent mapping across queries.
5. Alternative Approaches:
 - o BeanPropertyRowMapper: Spring provides a default RowMapper implementation called BeanPropertyRowMapper, which automatically maps ResultSet columns to JavaBean properties based on naming conventions (column names must match property names).



```
List<User> users = jdbcTemplate.query("SELECT * FROM users", new BeanPropertyRowMapper<>(User.class));
```

- Lambda Expressions: For simple mappings, you can use a lambda expression instead of a full RowMapper implementation:

```
List<User> users = jdbcTemplate.query("SELECT * FROM users", (rs, rowNum) -> new User(rs.getInt("id"), rs.getString("name"), rs.getString("email")));
```

6. When to Use:

- Use custom RowMapper when you need specific mapping logic or when column names don't match JavaBean properties.
- Use BeanPropertyRowMapper for straightforward mappings to save time and reduce code.

In summary, row mapping in Spring JDBC is a powerful mechanism to transform database query results into Java objects, making data access more intuitive and maintainable.

Using NamedParameterJdbcTemplate:

The **NamedParameterJdbcTemplate** in Spring JDBC is an extension of the JdbcTemplate class that provides a more convenient way to handle SQL queries by using **named parameters** instead of traditional ? placeholders. This makes queries more readable, maintainable, and less error-prone, especially for complex SQL statements with many parameters.

Key Features of NamedParameterJdbcTemplate

1. Named Parameters:

- Instead of using positional ? placeholders (e.g., SELECT * FROM users WHERE id = ? AND name = ?), you use named parameters with a : prefix (e.g., :id, :name).
- Example SQL:

```
SELECT * FROM users WHERE id = :id AND name = :name
```

2. Improved Readability:

- Named parameters make it clear what each parameter represents, reducing confusion in queries with multiple parameters.

3. Flexible Parameter Binding:

- Parameters can be provided as a Map, SqlParameterSource, or directly as method arguments, offering flexibility in how you pass values.

4. Built on JdbcTemplate:

- NamedParameterJdbcTemplate wraps a JdbcTemplate and delegates the actual database operations to it, so it supports all the features of JdbcTemplate (e.g., row mapping, exception handling).

How to Use NamedParameterJdbcTemplate

1. Creating an Instance:

- You need a DataSource to initialize NamedParameterJdbcTemplate.
- Example:

```
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import javax.sql.DataSource;
```

DataSource dataSource = // ... configure your DataSource



```
NamedParameterJdbcTemplate namedParameterJdbcTemplate = new
NamedParameterJdbcTemplate(dataSource);
```

2. Executing Queries:

- You can execute queries using methods like query(), queryForObject(), update(), etc., similar to JdbcTemplate, but with named parameters.

3. Binding Parameters:

- Parameters can be passed in several ways:

▪ Using a Map:

```
import java.util.HashMap;
import java.util.Map;
```

```
Map<String, Object> params = new HashMap<>();
params.put("id", 1);
params.put("name", "John Doe");
String sql = "SELECT * FROM users WHERE id = :id AND name = :name";
List<User> users = namedParameterJdbcTemplate.query(sql, params, new UserRowMapper());
```

▪ Using SqlParameterSource:

- Spring provides MapSqlParameterSource and BeanPropertySqlParameterSource for structured parameter passing.
- Example with MapSqlParameterSource:

```
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
```

```
MapSqlParameterSource params = new MapSqlParameterSource()
.addValue("id", 1)
.addValue("name", "John Doe");
String sql = "SELECT * FROM users WHERE id = :id AND name = :name";
List<User> users = namedParameterJdbcTemplate.query(sql, params, new UserRowMapper());
```

- Example with BeanPropertySqlParameterSource:

```
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
```

```
User user = new User(1, "John Doe", "john@example.com");
BeanPropertySqlParameterSource params = new BeanPropertySqlParameterSource(user);
String sql = "SELECT * FROM users WHERE id = :id AND name = :name";
List<User> users = namedParameterJdbcTemplate.query(sql, params, new UserRowMapper());
```

4. Row Mapping:

- Just like JdbcTemplate, you can use a RowMapper or BeanPropertyRowMapper to map query results to Java objects.
- Example:

```
String sql = "SELECT * FROM users WHERE id = :id";
Map<String, Object> params = new HashMap<>();
params.put("id", 1);
```



```
User user = namedParameterJdbcTemplate.queryForObject(sql, params, new BeanPropertyRowMapper<>(User.class));
```

5. Update Operations:

- For INSERT, UPDATE, or DELETE queries, use the update() method.
- Example:

```
String sql = "INSERT INTO users (id, name, email) VALUES (:id, :name, :email)";  
MapSqlParameterSource params = new MapSqlParameterSource()  
    .addValue("id", 2)  
    .addValue("name", "Jane Doe")  
    .addValue("email", "jane@example.com");  
int rowsAffected = namedParameterJdbcTemplate.update(sql, params);
```

Advantages of NamedParameterJdbcTemplate

- **Clarity:** Named parameters make SQL queries self-documenting and easier to understand.
- **Maintainability:** You don't need to worry about the order of parameters, unlike with ? placeholders.
- **Flexibility:** Supports multiple ways to bind parameters (Map, SqlParameterSource, etc.).
- **Reusability:** Named parameters allow you to reuse the same parameter multiple times in a query (e.g., :name can appear multiple times).

Limitations

- Slightly more verbose than JdbcTemplate for simple queries with few parameters.
- Requires additional configuration for parameter binding compared to positional placeholders.

When to Use

- Use NamedParameterJdbcTemplate when your SQL queries have multiple parameters, and you want to improve readability and maintainability.
- It's particularly useful for complex queries or when working with dynamic SQL where the parameter order might change.

Demo Application to demonstrate the above concepts:

Project Description: Employee Management System

The **Employee Management System** is a web-based application built using Spring Core, Thymeleaf, and MySQL, deployed on Tomcat 11 with Java 21. It provides a user-friendly interface for managing employee records, including adding, editing, deleting, viewing details, and batch importing employees. The application uses a MySQL database (employee_db) to store employee data and includes a view (employee_summary) for aggregated details. The system supports CRUD operations and batch processing, with a clean, responsive UI styled using CSS. Logging is configured via Logback, and the application is structured to ensure modularity and maintainability.

Key Features:

- List all employees with options to edit, delete, or view details.
- Add individual employees via a form.
- Edit existing employee records.
- View detailed employee information using a stored procedure.
- Batch import multiple employees through a single form.



- Display success/error messages for user actions.
- Consistent styling across all pages with context-relative resource loading.

Architecture:

- **Framework:** Spring Core (MVC) for handling requests and business logic.
- **Frontend:** Thymeleaf templates for dynamic HTML rendering.
- **Database:** MySQL with NamedParameterJdbcTemplate for data access.
- **Deployment:** Tomcat 11, with the application running under the /employee-management context path.
- **Logging:** Logback for debugging and error tracking.

Role of Each File

1. **Employee.java** (src/main/java/com/example/model/Employee.java)
 - Defines the Employee model with fields: id, firstName, lastName, email, department.
 - Provides getters and setters for data binding and database operations.
2. **EmployeeBatchForm.java** (src/main/java/com/example/model/EmployeeBatchForm.java)
 - A wrapper class holding a List<Employee> for binding multiple employees in the batch save form.
 - Facilitates Thymeleaf form binding for batch operations.
3. **EmployeeService.java** (src/main/java/com/example/service/EmployeeService.java)
 - Interface defining service-layer methods for CRUD and batch operations (save, update, delete, findAll, findById, getEmployeeDetailsViaStoredProcedure, saveBatch).
4. **EmployeeServiceImpl.java** (src/main/java/com/example/service/EmployeeServiceImpl.java)
 - Implements EmployeeService, delegating data operations to EmployeeDAO.
 - Provides business logic for managing employee data.
5. **EmployeeDAO.java** (src/main/java/com/example/dao/EmployeeDAO.java)
 - Interface defining data access methods for interacting with the employees table and employee_summary view.
6. **EmployeeDAOImpl.java** (src/main/java/com/example/dao/EmployeeDAOImpl.java)
 - Implements EmployeeDAO using NamedParameterJdbcTemplate for database operations.
 - Handles SQL queries for CRUD, batch inserts, and stored procedure calls.
7. **EmployeeController.java** (src/main/java/com/example/controller/EmployeeController.java)
 - Handles HTTP requests and responses, mapping endpoints to Thymeleaf templates.
 - Manages form submissions, redirects, and error/success messages using RedirectAttributes.
8. **DatabaseConfig.java** (src/main/java/com/example/config/DatabaseConfig.java)
 - Configures the MySQL data source and NamedParameterJdbcTemplate for database connectivity.
9. **WebConfig.java** (src/main/java/com/example/config/WebConfig.java)
 - Configures Spring MVC, Thymeleaf (ClassLoaderTemplateResolver, SpringTemplateEngine, ThymeleafViewResolver), and resource handlers for static files (/resources/**).
10. **employee-list.html** (src/main/resources/templates/employee-list.html)
 - Displays a table of all employees with columns for ID, full name, email, department, and action links (Edit, Delete, Details).
 - Includes links to add a single employee or batch employees.
 - Shows success/error messages after operations.
11. **employee-form.html** (src/main/resources/templates/employee-form.html)



- Provides a form for adding or editing a single employee.
 - Dynamically displays “Add Employee” or “Edit Employee” based on the presence of an employee ID.
 - Includes fields for firstName, lastName, email, department, with validation and a “Back to List” link.
12. **employee-details.html** (src/main/resources/templates/employee-details.html)
- Displays detailed information for a single employee (ID, first name, last name, email, department).
 - Includes a “Back to List” link.
13. **employee-batch-form.html** (src/main/resources/templates/employee-batch-form.html)
- Allows input of multiple employees (default: three) for batch saving.
 - Uses a div-based form with fields for each employee’s firstName, lastName, email, department.
 - Displays success/error messages and includes a “Back to List” link.
14. **style.css** (src/main/webapp/resources/css/style.css)
- Provides styling for all templates, including layout (.container), tables, forms (.form-group), buttons (.btn), and alerts (.alert.success, .alert.error).
 - Ensures a consistent, responsive design across pages.
15. **logback.xml** (src/main/resources/logback.xml)
- Configures Logback for logging to logs/catalina.out.
 - Sets debug level for Spring and info level for other logs, with a pattern for timestamp, thread, and message.
16. **schema.sql** (src/main/resources/schema.sql)
- Defines the database schema for employee_db, including the employees table and employee_summary view.
 - Includes optional sample data for testing.
17. **pom.xml** (pom.xml)
- Defines project dependencies (Spring Web MVC, Thymeleaf, MySQL Connector, Logback) and build configuration.
 - Sets the artifact ID to employee-management for WAR file generation.
18. **web.xml** (src/main/webapp/WEB-INF/web.xml)
- Configures the Spring DispatcherServlet for handling requests.
 - Maps the servlet to / for all application endpoints.

Endpoints

The EmployeeController.java defines the following endpoints, all under the /employee-management context path:

1. **GET /:**
 - Displays the employee list page (employee-list.html).
 - URL: http://localhost:8080/employee-management/
2. **GET /employee/new:**
 - Shows a form to add a new employee (employee-form.html).
 - URL: http://localhost:8080/employee-management/employee/new
3. **POST /employee/save:**
 - Saves a new employee or updates an existing one based on the presence of an ID.
 - Redirects to / with a success/error message.
 - URL: http://localhost:8080/employee-management/employee/save
4. **GET /employee/edit/{id}:**



- Displays a form pre-populated with an employee's data for editing (employee-form.html).
 - URL: <http://localhost:8080/employee-management/employee/edit/{id}> (e.g., /employee/edit/1)
5. **GET /employee/delete/{id}:**
- Deletes an employee by ID and redirects to / with a success/error message.
 - URL: <http://localhost:8080/employee-management/employee/delete/{id}> (e.g., /employee/delete/1)
6. **GET /employee/details/{id}:**
- Shows detailed employee information using a stored procedure (employee-details.html).
 - URL: <http://localhost:8080/employee-management/employee/details/{id}> (e.g., /employee/details/1)
7. **GET /employee/batch:**
- Displays a form to input multiple employees for batch saving (employee-batch-form.html).
 - URL: <http://localhost:8080/employee-management/employee/batch>
8. **POST /employee/batchSave:**
- Processes the batch form submission, saving valid employees and redirecting to / with a success/error message.
 - URL: <http://localhost:8080/employee-management/employee/batchSave>

Notes

- **Deployment:** The application is packaged as employee-management.war and deployed on Tomcat 11 at <http://localhost:8080/employee-management/>.
- **Database:** Assumes a MySQL database (employee_db) with the employees table and employee_summary view.
- **CSS:** Static resources are served from /resources/** (mapped to src/main/webapp/resources/) via WebConfig.java.
- **Error Handling:** All endpoints use RedirectAttributes for user feedback (success/error messages).
- **Testing:** Ensure the database is initialized with schema.sql, and monitor logs/catalina.out for debugging.

Transaction Management

Let's see, How the transection happens using the JDBC API.

A **transaction** is a sequence of one or more SQL operations treated as a single unit of work. It ensures data integrity by adhering to the **ACID** properties:

- **Atomicity:** All operations in a transaction are completed successfully, or none are applied.
- **Consistency:** The database remains in a consistent state before and after the transaction.
- **Isolation:** Transactions are isolated from each other until complete.
- **Durability:** Committed transactions are permanently saved, even in case of system failure.

In the context of the **JDBC (Java Database Connectivity) API**, transactions allow you to execute multiple SQL statements as a single unit, ensuring that either all statements succeed or none are applied.



Handling Transactions Using JDBC API

JDBC provides mechanisms to manage transactions explicitly. By default, JDBC operates in **auto-commit mode**, where each SQL statement is treated as a separate transaction and committed immediately. To handle transactions manually, you disable auto-commit and use explicit commit or rollback operations.

Here's how transactions are handled using the JDBC API:

1. Disable Auto-Commit Mode:

- By default, Connection objects in JDBC have auto-commit enabled (`setAutoCommit(true)`).
- To manage a transaction manually, disable auto-commit by calling: `connection.setAutoCommit(false);`

2. Execute SQL Statements:

- Perform the desired SQL operations (e.g., INSERT, UPDATE, DELETE) using Statement, PreparedStatement, or CallableStatement objects.
- These operations are not committed to the database until explicitly instructed.

3. Commit the Transaction:

- If all operations succeed, commit the transaction to make changes permanent:

```
connection.commit();
```

4. Rollback on Failure:

- If an error occurs, roll back the transaction to undo all changes:

```
connection.rollback();
```

- This ensures the database remains consistent.

5. Restore Auto-Commit (Optional):

- After completing the transaction, you can re-enable auto-commit if needed:

```
connection.setAutoCommit(true);
```

6. Use Savepoints (Optional):

- For partial rollbacks, you can set **savepoints** within a transaction:

```
Savepoint savepoint = connection.setSavepoint("SavepointName");
```

- Roll back to a specific savepoint if needed:

```
connection.rollback(savepoint);
```

Example: Handling a Transaction in JDBC

Here's a sample Java code demonstrating transaction management using JDBC:

```
import java.sql.*;  
  
public class TransactionExample {  
    public static void main(String[] args) {  
        String url = "jdbc:mysql://localhost:3306/mydb";  
        // JDBC code to establish connection, execute statements, etc.  
    }  
}
```



```
String user = "username";
String password = "password";

try (Connection connection = DriverManager.getConnection(url, user, password)) {
    // Disable auto-commit
    connection.setAutoCommit(false);

    try (PreparedStatement stmt1 = connection.prepareStatement(
        "UPDATE accounts SET balance = balance - ? WHERE account_id = ?");
        PreparedStatement stmt2 = connection.prepareStatement(
        "UPDATE accounts SET balance = balance + ? WHERE account_id = ?")) {

        // First operation: Deduct from sender's account
        stmt1.setDouble(1, 100.00);
        stmt1.setInt(2, 1);
        stmt1.executeUpdate();

        // Second operation: Add to receiver's account
        stmt2.setDouble(1, 100.00);
        stmt2.setInt(2, 2);
        stmt2.executeUpdate();

        // Commit the transaction
        connection.commit();
        System.out.println("Transaction committed successfully.");
    } catch (SQLException e) {
        // Rollback on error
        connection.rollback();
        System.out.println("Transaction rolled back due to error: " + e.getMessage());
    } finally {
        // Restore auto-commit
        connection.setAutoCommit(true);
    }
} catch (SQLException e) {
    System.out.println("Connection error: " + e.getMessage());
}
```

Explanation of the Example

- **Connection Setup:** Establishes a connection to a MySQL database.
- **Auto-Commit Disabled:** `setAutoCommit(false)` ensures manual transaction control.
- **SQL Operations:** Two UPDATE statements simulate a money transfer between accounts.



- **Commit or Rollback:** If both updates succeed, the transaction is committed. If an error occurs, the transaction is rolled back.
- **Resource Management:** The try-with-resources block ensures that Connection and PreparedStatement objects are closed automatically.
- **Error Handling:** Catches SQLException to handle database errors and rolls back the transaction if needed.

Key Points

- Always use try-catch blocks to handle exceptions and ensure proper rollback.
 - Close resources (e.g., Connection, Statement) properly, preferably using try-with-resources.
 - Savepoints are useful for complex transactions requiring partial rollbacks.
 - Ensure the database driver supports transactions (most modern databases like MySQL, PostgreSQL, etc., do).
-

Caching in Spring JDBC:

Spring JDBC, caching frequently accessed database results can significantly improve application performance by reducing the number of database queries, especially for data that doesn't change often. Spring's caching abstraction, available through the @Cacheable annotation, integrates seamlessly with Spring JDBC (part of the module) to cache query results. This allows you to store the results of expensive database operations in memory (using a cache like EhCache, Caffeine, or Redis) and reuse them for subsequent calls, avoiding redundant database hits.

1. Overview of Caching in Spring JDBC

- **Purpose:** Caching stores the results of database queries in memory to avoid repeated executions of the same query, improving performance for read-heavy operations.
- **Spring's Caching Abstraction:** Spring provides a declarative caching framework using annotations like @Cacheable, @CachePut, and @CacheEvict, which work with Spring JDBC's JdbcTemplate or other data access methods.
- **Key Annotation:** @Cacheable is used to mark methods whose results should be cached. When the method is called with the same input parameters, Spring returns the cached result instead of querying the database again.
- **Use Cases:**
 - Caching lookup tables or reference data that rarely changes (e.g., user roles, configuration settings).
 - Caching results of expensive queries (e.g., aggregations or joins).
 - Reducing database load in high-traffic applications.

2. How @Cacheable Works

- **Annotation:** @Cacheable is applied to a method to indicate that its return value should be cached.
- **Cache Key:** The cache key is generated based on the method's parameters (or a custom key if specified).



- **Cache Storage:** The result is stored in a cache (e.g., in-memory with Caffeine or distributed with Redis).
- **Behavior:**
 - If the cache contains a result for the given key, Spring returns it without executing the method.
 - If the cache misses, the method executes, the database is queried, and the result is stored in the cache for future use.
- **Integration with Spring JDBC:** When used with JdbcTemplate, @Cacheable caches the objects returned by queries (e.g., List, Map, or custom domain objects).

3. Setting Up Caching in Spring

To use @Cacheable with Spring JDBC, you need to:

1. Enable caching in your Spring application.
2. Configure a cache manager (e.g., ConcurrentMapCacheManager, CaffeineCacheManager, or RedisCacheManager).
3. Use @Cacheable on methods that perform database queries.

Step 1: Enable Caching

Add @EnableCaching to a configuration class to enable Spring's caching support.

```
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableCaching
public class CacheConfig {
    // Cache manager configuration goes here (see below)
}
```

Cache Providers:

1. ConcurrentMapCacheManager

- **Description:** A simple in-memory cache manager provided by Spring out of the box. It uses a ConcurrentHashMap as the underlying cache, suitable for small applications, testing, or development environments. Not recommended for production due to lack of eviction policies or persistence.
- **Use Case:** Lightweight caching for single-instance applications or prototyping.
- **Pros:** No external dependencies, easy to set up.
- **Cons:** No advanced features (e.g., expiration, persistence), not distributed.

Dependency

No additional dependency is required, as ConcurrentMapCacheManager is included in spring-context.

```
<!-- Already included in spring-context -->
<dependency>
    <groupId>org.springframework</groupId>
```



```
<artifactId>spring-context</artifactId>
<version>6.1.14</version>
</dependency>
```

Configuration

```
import org.springframework.cache.CacheManager;
import org.springframework.cache.concurrent.ConcurrentMapCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.cache.annotation.EnableCaching;
```

```
@Configuration
@EnableCaching
public class CacheConfig {
    @Bean
    public CacheManager cacheManager() {
        // Define cache names
        return new ConcurrentMapCacheManager("usersCache", "ordersCache");
    }
}
```

Explanation:

- Creates a ConcurrentMapCacheManager with two caches: usersCache and ordersCache.
- @EnableCaching enables Spring's caching support.
- Caches are stored in memory using ConcurrentHashMap.

2. Caffeine

- **Description:** A high-performance, in-memory caching library optimized for modern Java applications. It provides advanced features like expiration policies, size-based eviction, and statistics.
- **Use Case:** Production-grade in-memory caching for single-instance applications requiring efficient eviction and expiration.
- **Pros:** Fast, feature-rich (e.g., TTL, size limits), modern alternative to EhCache.
- **Cons:** Not distributed; limited to single JVM.

Dependency

Add the Caffeine dependency to your pom.xml:

```
<dependency>
    <groupId>com.github.ben-manes.caffeine</groupId>
    <artifactId>caffeine</artifactId>
    <version>3.1.8</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>6.1.14</version>
```



```
</dependency>
Configuration
import com.github.benmanes.caffeine.cache.Caffeine;
import org.springframework.cache.CacheManager;
import org.springframework.cache.caffeine.CaffeineCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.cache.annotation.EnableCaching;
import java.util.concurrent.TimeUnit;

@Configuration
@EnableCaching
public class CacheConfig {
    @Bean
    public CacheManager cacheManager() {
        CaffeineCacheManager cacheManager = new CaffeineCacheManager("usersCache",
"ordersCache");
        cacheManager.setCaffeine(Caffeine.newBuilder()
            .initialCapacity(100)
            .maximumSize(1000)
            .expireAfterAccess(10, TimeUnit.MINUTES)
            .recordStats());
        return cacheManager;
    }
}
```

Explanation:

- Configures a CaffeineCacheManager with two caches: usersCache and ordersCache.
- Caffeine settings:
 - initialCapacity(100): Initial size of the cache.
 - maximumSize(1000): Maximum number of entries.
 - expireAfterAccess(10, TimeUnit.MINUTES): Entries expire 10 minutes after last access.
 - recordStats(): Enables cache statistics for monitoring.

3. EhCache

- **Description:** A robust, widely-used in-memory caching library supporting advanced features like expiration, persistence, and clustering (in EhCache 3.x). Suitable for both standalone and distributed environments.
- **Use Case:** Production applications needing persistence, clustering, or advanced cache management.
- **Pros:** Feature-rich, supports disk persistence, can be clustered.
- **Cons:** More complex to configure than Caffeine, slightly slower.

Dependency

Add EhCache and Spring's cache support to your pom.xml:



```
<dependency>
    <groupId>org.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>3.10.8</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>6.1.14</version>
</dependency>
```

Configuration

Java-based Configuration:

```
import org.ehcache.config.builders.CacheConfigurationBuilder;
import org.ehcache.config.builders.ExpiryPolicyBuilder;
import org.ehcache.config.builders.ResourcePoolsBuilder;
import org.ehcache.jsr107.Eh107Configuration;
import org.springframework.cache.CacheManager;
import org.springframework.cache.jcache.JCacheCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.cache.annotation.EnableCaching;
import javax.cache.Caching;
import javax.cache.configuration.MutableConfiguration;
import javax.cache.expiry.Duration;
import javax.cache.expiry.TouchedExpiryPolicy;
import java.util.concurrent.TimeUnit;
```

```
@Configuration
@EnableCaching
public class CacheConfig {
    @Bean
    public CacheManager cacheManager() {
        // Create JCache (JSR-107) configuration
        MutableConfiguration<Object, Object> cacheConfig = new MutableConfiguration<>()
            .setTypes(Object.class, Object.class)
            .setStoreByValue(false)
            .setExpiryPolicyFactory(TouchedExpiryPolicy.factoryOf(new
Duration(TimeUnit.MINUTES, 10)));
        // Create JCache CacheManager
        javax.cache.CacheManager jCacheManager =
        Caching.getCacheProvider().getCacheManager();
        jCacheManager.createCache("usersCache", cacheConfig);
        jCacheManager.createCache("ordersCache", cacheConfig);
```



```
        return new JCacheCacheManager(jCacheManager);
    }
}
```

XML-based Configuration (Optional): Create an ehcache.xml file in src/main/resources:

```
<ehcache:config
  xmlns:ehcache="http://www.ehcache.org/v3"
  xmlns:jsr107="http://www.ehcache.org/v3/jsr107">
  <ehcache:cache alias="usersCache">
    <ehcache:expiry>
      <ehcache:ttl unit="minutes">10</ehcache:ttl>
    </ehcache:expiry>
    <ehcache:heap unit="entries">1000</ehcache:heap>
  </ehcache:cache>
  <ehcache:cache alias="ordersCache">
    <ehcache:expiry>
      <ehcache:ttl unit="minutes">10</ehcache:ttl>
    </ehcache:expiry>
    <ehcache:heap unit="entries">1000</ehcache:heap>
  </ehcache:cache>
</ehcache:config>
```

Then configure Spring to use it:

```
import org.springframework.cache.jcache.JCacheCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.cache.annotation.EnableCaching;
import javax.cache.Caching;
```

```
@Configuration
@EnableCaching
public class CacheConfig {
    @Bean
    public CacheManager cacheManager() {
        return new JCacheCacheManager(Caching.getCachingProvider().getCacheManager(
            getClass().getResource("/ehcache.xml").toURI(),
            getClass().getClassLoader()
        ));
    }
}
```

Explanation:

- Uses EhCache 3.x with JSR-107 (JCache) integration.
- Configures two caches (usersCache, ordersCache) with a 10-minute TTL and a maximum of 1000 entries.



- The XML configuration is optional for more complex setups (e.g., persistence or clustering).

Using @Cacheable, @CachePut, and @CacheEvict:

In Spring's caching abstraction, the annotations `@Cacheable`, `@CachePut`, and `@CacheEvict` are used to manage cached data declaratively, integrating seamlessly with Spring JDBC or other data access mechanisms. These annotations allow developers to control how data is stored, updated, or removed from a cache without writing explicit cache-handling code.

1. `@Cacheable`

- **Purpose:** Marks a method's return value to be cached. When the method is called with the same input parameters, Spring returns the cached result instead of executing the method again, reducing database queries or expensive computations.
- **Use Case:** Caching frequently accessed, relatively stable data, such as database query results (e.g., user details, lookup tables).
- **Behavior:**
 - On first invocation with a specific key, the method executes, and the result is stored in the cache.
 - Subsequent calls with the same key return the cached result without executing the method.
- **Key Attributes:**
 - `value` or `cacheNames`: Specifies the cache name(s) to store the result (e.g., "usersCache").
 - `key`: Defines the cache key, often using a SpEL expression (e.g., `#userId`).
 - `condition`: A SpEL expression to determine if the result should be cached (e.g., `#userId != null`).
 - `unless`: A SpEL expression to prevent caching if true (e.g., `#result == null`).
 - `sync`: Ensures thread-safe cache access (useful for concurrent scenarios).

Example with Spring JDBC

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;

@Service
public class UserService {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Cacheable(value = "usersCache", key = "#userId", unless = "#result == null")
    public User getUserById(String userId) {
        try {
            return jdbcTemplate.queryForObject(
```



```

        "SELECT id, name FROM users WHERE id = ?",
        new Object[]{userId},
        (rs, rowNum) -> new User(rs.getString("id"), rs.getString("name"))
    );
} catch (EmptyResultDataAccessException e) {
    return null; // Not cached due to unless="#result == null"
}
}
}
}
}

```

Explanation:

- **Annotation:** @Cacheable(value = "usersCache", key = "#userId", unless = "#result == null")
 - value = "usersCache": Stores results in a cache named usersCache.
 - key = "#userId": Uses the userId parameter as the cache key.
 - unless = "#result == null": Prevents caching if the result is null (e.g., no user found).
- **Behavior:**
 - First call to getUserById("123"): Executes the query, caches the User object with key "123".
 - Subsequent calls with userId = "123": Returns the cached User without querying the database.
- **Error Handling:** Catches EmptyResultDataAccessException to handle cases where no user is found, returning null.

2. @CachePut

- **Purpose:** Updates the cache with the method's return value every time the method is called. Unlike @Cacheable, the method always executes, ensuring the cache reflects the latest data.
- **Use Case:** Updating cached data after a database modification (e.g., updating a user's details).
- **Behavior:**
 - The method executes, performs the database operation, and the return value is stored in the cache with the specified key.
 - Does not check the cache before execution, ensuring the database and cache are updated.
- **Key Attributes:**
 - value or cacheNames: Specifies the cache name(s).
 - key: Defines the cache key (SpEL expression).
 - condition: A SpEL expression to determine if the cache should be updated.
 - unless: A SpEL expression to prevent cache updates if true.

Example with Spring JDBC

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.CachePut;
import org.springframework.jdbc.core.JdbcTemplate;

```



```

import org.springframework.stereotype.Service;

@Service
public class UserService {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @CachePut(value = "usersCache", key = "#userId")
    public User updateUser(String userId, String newName) {
        jdbcTemplate.update(
            "UPDATE users SET name = ? WHERE id = ?",
            newName, userId
        );
        return new User(userId, newName); // Cache this updated user
    }
}

```

Explanation:

- **Annotation:** @CachePut(value = "usersCache", key = "#userId")
 - Updates the usersCache with the key userId.
- **Behavior:**
 - The method updates the users table with the new name.
 - Returns a new User object, which is stored in the cache under the key userId.
 - Ensures the cache reflects the updated user data, so subsequent @Cacheable calls to getUserById return the latest value.
- **Use Case:** Used when modifying data to keep the cache in sync with the database.

3. @CacheEvict

- **Purpose:** Removes one or more entries from the cache, typically after a database operation that invalidates cached data (e.g., deleting or updating data).
- **Use Case:** Clearing cache entries when data is deleted or modified to prevent stale data.
- **Behavior:**
 - Removes the specified cache entry (or all entries) when the method is called.
 - The method executes its logic (e.g., database delete), and the cache is updated accordingly.
- **Key Attributes:**
 - value or cacheNames: Specifies the cache name(s).
 - key: Defines the cache key to evict (SpEL expression).
 - allEntries: If true, clears all entries in the specified cache (default: false).
 - beforeInvocation: If true, evicts the cache before the method executes (default: false).
 - condition: A SpEL expression to determine if eviction should occur.

Example with Spring JDBC



```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;

@Service
public class UserService {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @CacheEvict(value = "usersCache", key = "#userId")
    public void deleteUser(String userId) {
        jdbcTemplate.update("DELETE FROM users WHERE id = ?", userId);
    }
}

```

Explanation:

- Annotation:** `@CacheEvict(value = "usersCache", key = "#userId")`
 - Removes the cache entry with the key `userId` from `usersCache`.
- Behavior:**
 - Deletes the user from the database.
 - Removes the corresponding entry from the cache, ensuring no stale data remains.
- Use Case:** Used after deleting a user to prevent `@Cacheable` methods from returning outdated data.

Example with allEntries

```

@CacheEvict(value = "usersCache", allEntries = true)
public void resetAllUsers() {
    jdbcTemplate.update("TRUNCATE TABLE users");
}

```

- Behavior:** Clears all entries in `usersCache` after truncating the `users` table.

4. Key Differences Between `@Cacheable`, `@CachePut`, and `@CacheEvict`

Annotation	Purpose	Method Execution	Cache Effect	Use Case
<code>@Cacheable</code>	Cache method's return value	Skipped if cache hit	Stores result if cache miss	Fetching frequently accessed data
<code>@CachePut</code>	Update cache with method's return value	Always executed	Updates cache with new result	Updating data and keeping cache in sync
<code>@CacheEvict</code>	Remove cache entries	Always executed	Removes specified or all cache entries	Invalidating cache after data changes



Pagination And Sorting:

Pagination

Pagination is the process of dividing a large dataset into smaller, manageable chunks called **pages**. Instead of fetching all data at once, pagination retrieves a specific subset of records (e.g., 10 records per page) based on a page number and page size. This is critical for improving performance and user experience when dealing with large datasets, such as displaying search results or table data in a web application.

- **Key Parameters:**

- **Page Number:** Specifies which page to retrieve (e.g., page 1, page 2). Often zero-based in programming.
- **Page Size:** Number of records per page (e.g., 10 or 20).
- **Offset:** The starting point in the dataset, calculated as $\text{pageNumber} * \text{pageSize}$.
- **Limit:** The number of records to fetch (usually equal to page size).

Sorting

Sorting is the process of arranging data in a specific order based on one or more columns/fields. It's typically done using the ORDER BY clause in SQL, specifying the column and direction (ascending or descending).

- **Key Parameters:**

- **Sort By:** The column/field to sort on (e.g., name, created_date).
- **Sort Order:** The direction of sorting:
 - ASC (ascending, e.g., A-Z, 1-10).
 - DESC (descending, e.g., Z-A, 10-1).

Project Description: Spring 6 CRUD Application with Caching, Pagination, and Sorting

This project is a web-based CRUD (Create, Update, Read, Delete) application built using **Spring 6**, **MySQL**, **Thymeleaf**, and **EhCache**. It manages a list of users with basic attributes (ID and Name), providing a user-friendly interface to perform CRUD operations. The application incorporates **server-side caching** using EhCache to optimize data retrieval, **pagination** to display users in manageable chunks, and **sorting** to order users by ID or Name. The system is deployed on **Tomcat 11** with **Java 21**, ensuring compatibility with modern Java EE standards. Key features include:

- **CRUD Operations:** Users can create, read, update, and delete user records via a web interface.
- **Caching:** EhCache with JCache integration caches user data to reduce database queries, with cache eviction and updates handled appropriately.
- **Pagination:** Displays users in pages with configurable page sizes, improving performance for large datasets.
- **Sorting:** Allows sorting users by ID or Name in ascending or descending order.
- **Transaction Management:** Ensures data consistency with Spring's @Transactional annotations.
- **Error Handling:** Displays user-friendly error messages for invalid inputs or operations.

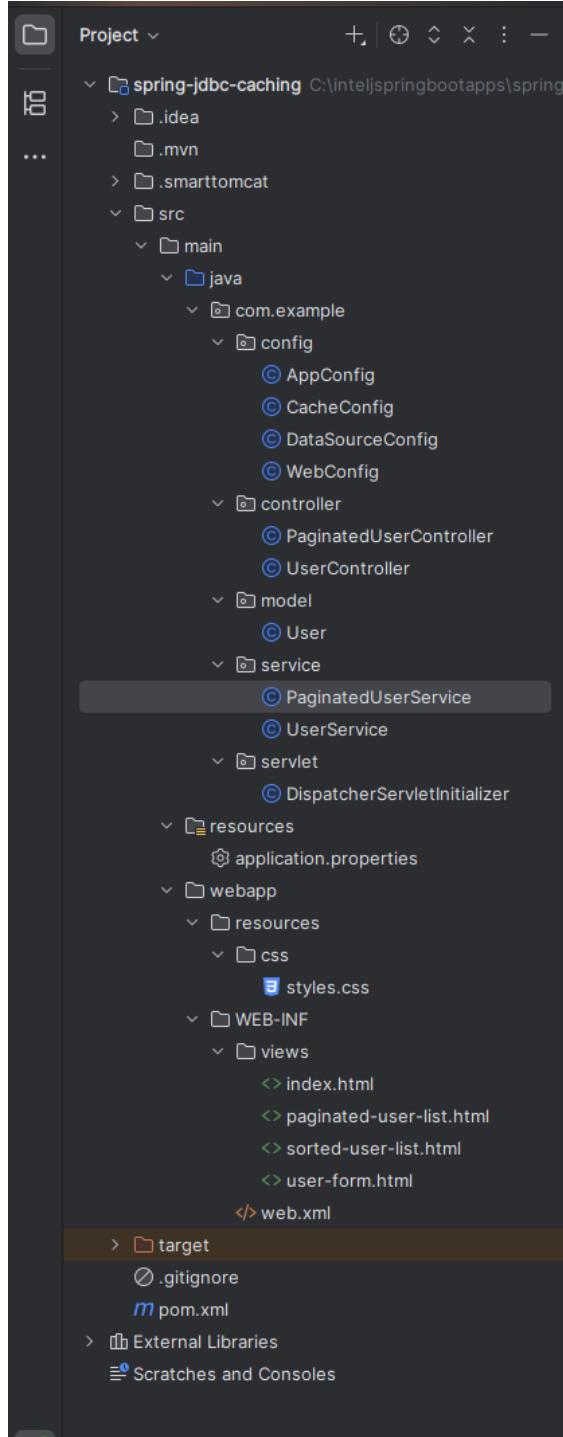


- **Responsive UI:** Uses Thymeleaf templates with CSS styling for a clean, modern interface.

The application is designed to be extensible, with separate services and controllers for pagination and sorting to avoid modifying core CRUD functionality. It maintains cache consistency across operations and leverages server-side pagination and sorting for efficiency.

Project Folder Structure and File Roles

Below is the project folder structure with each file's role, reflecting the implementation of the CRUD application with caching, pagination, and sorting.



File Descriptions and Roles

- **src/main/java/com/example/config/**
 - **AppConfig.java:** Main Spring configuration class that enables component scanning and imports other configuration classes (DataSourceConfig, CacheConfig, WebConfig) to bootstrap the application.

```
• package com.example.config;

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

@Configuration
@Import({DataSourceConfig.class, CacheConfig.class, WebConfig.class})
public class AppConfig {
```

- **CacheConfig.java:** Configures EhCache with JCache as the caching provider, defining the usersCache cache and enabling transaction-aware caching to synchronize cache operations with database transactions.

```
• package com.example.config;

import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.cache.jcache.JCacheCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.cache.Caching;
import javax.cache.configuration.MutableConfiguration;
import javax.cache.expiry.Duration;
import javax.cache.expiry.TouchedExpiryPolicy;
import java.util.concurrent.TimeUnit;

@Configuration
@EnableCaching
public class CacheConfig {

    @Bean
    public CacheManager cacheManager() {
        // Configure cache properties
        MutableConfiguration<Object, Object> cacheConfig = new
        MutableConfiguration<>()
            .setTypes(Object.class, Object.class)
            .setStoreByValue(false)

        .setExpiryPolicyFactory(TouchedExpiryPolicy.factoryOf(new
        Duration(TimeUnit.MINUTES, 1)));

        // Get JCache CacheManager
        javax.cache.CacheManager jCacheManager =
        Caching.getCachingProvider().getCacheManager();

        // Create cache only if it doesn't exist
        if (jCacheManager.getCache("usersCache") == null) {
            jCacheManager.createCache("usersCache", cacheConfig);
        }
    }
}
```



```
        return new JCacheCacheManager(jCacheManager);
    }
}
```

- **DataSourceConfig.java:** Configures the MySQL data source using HikariCP for connection pooling and provides a JdbcTemplate bean for database operations.
- package com.example.config;
- ```
import com.zaxxer.hikari.HikariDataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import
org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.PlatformTransactionManager;
import
org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;
import java.util.Properties;

@Configuration
@EnableTransactionManagement
public class DataSourceConfig {

 @Bean
 public DataSource dataSource() {
 HikariDataSource dataSource = new HikariDataSource();
 Properties properties = new Properties();
 try {

properties.load(getClass().getClassLoader().getResourceAsStream("appl
ication.properties"));
 } catch (Exception e) {
 throw new RuntimeException("Failed to load
application.properties", e);
 }
 dataSource.setJdbcUrl(properties.getProperty("db.url"));

 dataSource.setUsername(properties.getProperty("db.username"));

 dataSource.setPassword(properties.getProperty("db.password"));

 dataSource.setDriverClassName(properties.getProperty("db.driver"));
 return dataSource;
 }

 @Bean
 public JdbcTemplate jdbcTemplate(DataSource dataSource) {
 return new JdbcTemplate(dataSource);
 }

 @Bean
 public PlatformTransactionManager transactionManager(DataSource
dataSource) {
 return new DataSourceTransactionManager(dataSource);
 }
}
```



```
 }
}
```

- **DispatcherServletInitializer.java:** Initializes the Spring MVC DispatcherServlet, replacing the traditional web.xml for servlet configuration in a Java-based setup.
- ```
package com.example.servlet;

import com.example.config.AppConfig;
import com.example.config.CacheConfig;
import com.example.config.DataSourceConfig;
import com.example.config.WebConfig;
import
org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class DispatcherServletInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[]{AppConfig.class, CacheConfig.class,
DataSourceConfig.class};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[]{WebConfig.class};
    }

    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }
}
```

- **WebConfig.java:** Configures Spring MVC, including view resolution for Thymeleaf templates and static resource handling (e.g., CSS files).
- ```
package com.example.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import
org.springframework.web.servlet.config.annotation.EnableWebMvc;
import
org.springframework.web.servlet.config.annotation.ResourceHandlerRegi
stry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.thymeleaf.spring6.SpringTemplateEngine;
import
org.thymeleaf.spring6.templateresolver.SpringResourceTemplateResolver
;
import org.thymeleaf.spring6.view.ThymeleafViewResolver;
import org.thymeleaf.templatemode.TemplateMode;

@Configuration
```



```

@EnableWebMvc
@ComponentScan(basePackages = "com.example")
public class WebConfig implements WebMvcConfigurer {

 @Bean
 public SpringResourceTemplateResolver templateResolver() {
 SpringResourceTemplateResolver resolver = new
 SpringResourceTemplateResolver();
 resolver.setPrefix("/WEB-INF/views/");
 resolver.setSuffix(".html");
 resolver.setTemplateMode(TemplateMode.HTML);
 resolver.setCacheable(false); // Disable for development
 return resolver;
 }

 @Bean
 public SpringTemplateEngine templateEngine() {
 SpringTemplateEngine engine = new SpringTemplateEngine();
 engine.setTemplateResolver(templateResolver());
 return engine;
 }

 @Bean
 public ThymeleafViewResolver viewResolver() {
 ThymeleafViewResolver resolver = new ThymeleafViewResolver();
 resolver.setTemplateEngine(templateEngine());
 resolver.setCharacterEncoding("UTF-8");
 return resolver;
 }

 @Override
 public void addResourceHandlers(ResourceHandlerRegistry registry)
 {
 registry.addResourceHandler("/resources/**")
 .addResourceLocations("/resources/");
 }
}

```

- **src/main/java/com/example/controller/**
  - **PaginatedUserController.java:** Handles requests for paginated and sorted user lists, mapping /paginated-user-list and /sorted-user-list to their respective Thymeleaf templates. Passes paginated or sorted data and parameters (e.g., page, size, sortBy, sortDir) to the views.

- package com.example.controller;

```

import com.example.service.PaginatedUserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class PaginatedUserController {

 @Autowired
 private PaginatedUserService paginatedUserService;

 @GetMapping("/paginated-user-list")

```



```

 public String paginatedList(Model model,
 @RequestParam(defaultValue = "1") int
page,
 @RequestParam(defaultValue = "5") int
size) {
 model.addAttribute("users",
paginatedUserService.getPaginatedUsers(page, size));
 model.addAttribute("currentPage", page);
 model.addAttribute("pageSize", size);
 model.addAttribute("totalUsers",
paginatedUserService.getTotalUsers());
 return "paginated-user-list";
}

@GetMapping("/sorted-user-list")
public String sortedList(Model model,
 @RequestParam(defaultValue = "id")
String sortBy,
 @RequestParam(defaultValue = "asc")
String sortDir) {
 model.addAttribute("users",
paginatedUserService.getSortedUsers(sortBy, sortDir));
 model.addAttribute("sortBy", sortBy);
 model.addAttribute("sortDir", sortDir);
 return "sorted-user-list";
}
}

```

- **UserController.java:** Manages core CRUD operations, mapping endpoints like / (user list), /user/add (create), /user/edit/{id} (update), and /user/delete/{id} (delete) to Thymeleaf templates. Handles error messages for invalid inputs.
  - package com.example.controller;
- ```

import com.example.model.User;
import com.example.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import
org.springframework.web.servlet.mvc.support.RedirectAttributes;

@Controller
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("/")
    public String index(Model model) {
        model.addAttribute("users", userService.getAllUsers());
        return "index";
    }

    @GetMapping("/user/add")

```



```
public String showAddForm(Model model) {
    model.addAttribute("user", new User());
    return "user-form";
}

@PostMapping("/user/add")
public String addUser(@ModelAttribute User user,
RedirectAttributes redirectAttributes) {
    try {
        userService.createUser(user.getId(), user.getName());
        return "redirect:/";
    } catch (IllegalArgumentException e) {
        redirectAttributes.addFlashAttribute("errorMessage",
e.getMessage());
        return "redirect:/user/add";
    }
}

@GetMapping("/user/edit/{id}")
public String showEditForm(@PathVariable("id") String userId,
Model model) {
    try {
        User user = userService.getUserById(userId);
        model.addAttribute("user", user != null ? user : new
User());
        return "user-form";
    } catch (IllegalArgumentException e) {
        model.addAttribute("errorMessage", e.getMessage());
        model.addAttribute("user", new User());
        return "user-form";
    }
}

@PostMapping("/user/edit/{id}")
public String updateUser(@PathVariable("id") String userId,
@ModelAttribute User user, RedirectAttributes redirectAttributes) {
    try {
        userService.updateUser(userId, user.getName());
        return "redirect:/";
    } catch (IllegalArgumentException e) {
        redirectAttributes.addFlashAttribute("errorMessage",
e.getMessage());
        return "redirect:/user/edit/" + userId;
    }
}

@GetMapping("/user/delete/{id}")
public String deleteUser(@PathVariable("id") String userId,
RedirectAttributes redirectAttributes) {
    try {
        userService.deleteUser(userId);
        return "redirect:/";
    } catch (IllegalArgumentException e) {
        redirectAttributes.addFlashAttribute("errorMessage",
e.getMessage());
        return "redirect:/";
    }
}

@ExceptionHandler(Exception.class)
public String handleException(Exception e, Model model) {
```



```

        model.addAttribute("errorMessage", "An error occurred: " +
e.getMessage());
        model.addAttribute("users", userService.getAllUsers());
        return "index";
    }
}

```

- **src/main/java/com/example/model/**

- **User.java:** Represents the User entity with id and name attributes, used for data transfer between the database and the application.

- package com.example.model;

```

public class User {
    private String id;
    private String name;

    public User() {
    }

    public User(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public String getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

```

- **src/main/java/com/example/service/**

- **PaginatedUserService.java:** Provides methods for server-side pagination (getPaginatedUsers) and sorting (getSortedUsers), using JdbcTemplate with SQL LIMIT, OFFSET, and ORDER BY. Caches results in usersCache with unique keys to avoid conflicts with UserService.

- package com.example.service;

```

import com.example.model.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;

import java.util.List;

@Service

```



```

public class PaginatedUserService {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Cacheable(value = "usersCache", key = "'paginatedUsers_page' +
#page + '_size' + #size")
    public List<User> getPaginatedUsers(int page, int size) {
        int offset = (page - 1) * size;
        return jdbcTemplate.query(
            "SELECT id, name FROM users LIMIT ? OFFSET ?",
            new Object[]{size, offset},
            (rs, rowNum) -> new User(rs.getString("id"),
            rs.getString("name"))
        );
    }

    @Cacheable(value = "usersCache", key = "'sortedUsers_sortBy' +
#sortBy + '_sortDir' + #sortDir")
    public List<User> getSortedUsers(String sortBy, String sortDir) {
        String column = "id".equals(sortBy) ? "id" : "name";
        String direction = "desc".equals(sortDir) ? "DESC" : "ASC";
        String query = String.format("SELECT id, name FROM users
ORDER BY %s %s", column, direction);
        return jdbcTemplate.query(
            query,
            (rs, rowNum) -> new User(rs.getString("id"),
            rs.getString("name"))
        );
    }

    public int getTotalUsers() {
        return jdbcTemplate.queryForObject("SELECT COUNT(*) FROM
users", Integer.class);
    }
}

```

- **UserService.java:** Handles core CRUD operations (getUserById, getAllUsers, createUser, updateUser, deleteUser) using JdbcTemplate. Applies caching with @Cacheable, @CachePut, and @CacheEvict to optimize data retrieval and ensure cache consistency.

```

• package com.example.service;

import com.example.model.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.cache.annotation.CachePut;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.dao.EmptyResultDataAccessException;

import java.util.List;
import java.util.Objects;

@Service
public class UserService {

```



```

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Cacheable(value = "usersCache", key = "#userId", unless =
    "#result == null")
    public User getUserById(String userId) {
        Objects.requireNonNull(userId, "userId cannot be null");
        try {
            return jdbcTemplate.queryForObject(
                "SELECT id, name FROM users WHERE id = ?",
                new Object[]{userId},
                (rs, rowNum) -> new User(rs.getString("id"),
                rs.getString("name"))
            );
        } catch (EmptyResultDataAccessException e) {
            return null;
        }
    }

    @Transactional
    @CachePut(value = "usersCache", key = "#userId")
    public User updateUser(String userId, String name) {
        Objects.requireNonNull(userId, "userId cannot be null");
        Objects.requireNonNull(name, "name cannot be null");
        jdbcTemplate.update("UPDATE users SET name = ? WHERE id = ?",
        name, userId);
        return new User(userId, name);
    }

    @Transactional
    @CacheEvict(value = "usersCache", key = "#userId")
    public void deleteUser(String userId) {
        Objects.requireNonNull(userId, "userId cannot be null");
        jdbcTemplate.update("DELETE FROM users WHERE id = ?",
        userId);
    }

    @Transactional
    @CacheEvict(value = "usersCache", allEntries = true)
    public void createUser(String userId, String name) {
        Objects.requireNonNull(userId, "userId cannot be null");
        Objects.requireNonNull(name, "name cannot be null");
        jdbcTemplate.update("INSERT INTO users (id, name) VALUES (?, ?)",
        userId, name);
    }

    @Cacheable(value = "usersCache", key = "'allUsers'")
    public List<User> getAllUsers() {
        return jdbcTemplate.query(
            "SELECT id, name FROM users",
            (rs, rowNum) -> new User(rs.getString("id"),
            rs.getString("name"))
        );
    }
}

```

- ***src/main/resources/***



- **application.properties**: Configures application properties, including MySQL database connection details (e.g., URL, username, password) for the data source.
- db.url=jdbc:mysql://localhost:3306/spring_cache_db?useSSL=false&serverTimezone=UTC
db.username=root
db.password=Archer@12345
db.driver=com.mysql.cj.jdbc.Driver

- **logback.xml**: Configures SLF4J with Logback for logging application events, including cache operations and database queries (optional, for debugging).
- **src/main/webapp/resources/css/**
 - **styles.css**: Defines CSS styles for all Thymeleaf templates, including table layouts, buttons (Add, Edit, Delete, navigation), pagination controls, sorting dropdowns, and error messages. Ensures a consistent, responsive UI.

```
```css
body {
 font-family: Arial, sans-serif;
 margin: 0;
 padding: 20px;
 background-color: #f4f4f9;
}

.container {
 max-width: 800px;
 margin: 0 auto;
 background: #fff;
 padding: 20px;
 border-radius: 8px;
 box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}

h1 {
 color: #333;
 text-align: center;
}

.error-message {
 color: #f44336;
 background-color: #ffebee;
 padding: 10px;
 margin-bottom: 15px;
 border-radius: 4px;
}

table {
 width: 100%;
 border-collapse: collapse;
 margin-top: 20px;
}

th, td {
 padding: 12px;
 text-align: left;
 border-bottom: 1px solid #ddd;
}
```



```
th {
 background-color: #4CAF50;
 color: white;
}

tr:hover {
 background-color: #f5f5f5;
}

.btn {
 display: inline-block;
 padding: 8px 12px;
 text-decoration: none;
 border-radius: 4px;
 font-size: 14px;
}

.btn-add {
 background-color: #4CAF50;
 color: white;
 margin-bottom: 20px;
}

.btn-add:hover {
 background-color: #45a049;
}

.btn-edit {
 background-color: #2196F3;
 color: white;
}

.btn-edit:hover {
 background-color: #1e88e5;
}

.btn-delete {
 background-color: #f44336;
 color: white;
}

.btn-delete:hover {
 background-color: #d32f2f;
}

.btn-nav {
 background-color: #666;
 color: white;
 margin-right: 10px;
}

.btn-nav:hover {
 background-color: #555;
}

.btn-nav.active {
 background-color: #4CAF50;
 font-weight: bold;
}

.pagination {
```



```
 margin: 20px 0;
 display: flex;
 align-items: center;
 gap: 10px;
 }

.pagination select {
 padding: 8px;
 border-radius: 4px;
 border: 1px solid #ddd;
}

.sorting {
 margin: 20px 0;
 display: flex;
 align-items: center;
 gap: 10px;
}

.sorting select {
 padding: 8px;
 border-radius: 4px;
 border: 1px solid #ddd;
}

form {
 margin-top: 20px;
}

label {
 display: inline-block;
 width: 100px;
 font-weight: bold;
}

input[type="text"] {
 padding: 8px;
 width: 200px;
 margin-bottom: 10px;
 border: 1px solid #ddd;
 border-radius: 4px;
}

button {
 padding: 8px 16px;
 background-color: #4CAF50;
 color: white;
 border: none;
 border-radius: 4px;
 cursor: pointer;
}

button:hover {
 background-color: #45a049;
}

a.cancel {
 color: #f44336;
 text-decoration: none;
 margin-left: 10px;
}
```



```
a.cancel:hover {
 text-decoration: underline;
}
```

- **src/main/webapp/WEB-INF/views/**

- **index.html:** The main user list page, displaying all users in a table with Add, Edit, and Delete actions. Includes navigation links to paginated-user-list.html and sorted-user-list.html. Uses cached data from UserService.getAllUsers.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <title>User Management</title>
 <meta charset="UTF-8"/>
 <link rel="stylesheet" th:href="@{/resources/css/styles.css}"/>
</head>
<body>
<div class="container">
 <h1>User Management</h1>
 <h3>Initially due to @cacheable, reads all data from the database, as duration set to 1 Min, next read before one minute give the same data so edit within 1 min not reflected. it will reflect reload after the 1 min</h3>
 <div th:if="${errorMessage}" class="error-message">
 th:text="${errorMessage}"></div>
 <a th:href="@{/user/add}" class="btn btn-add">Add New User
 <a th:href="@{/paginated-user-list}" class="btn btn-nav">View Paginated List
 <a th:href="@{/sorted-user-list}" class="btn btn-nav">View Sorted List
 <table>
 <thead>
 <tr>
 <th>ID</th>
 <th>Name</th>
 <th>Actions</th>
 </tr>
 </thead>
 <tbody>
 <tr th:each="user : ${users}">
 <td th:text="${user.id}"></td>
 <td th:text="${user.name}"></td>
 <td>
 <a th:href="@{/user/edit/{id} (id=${user.id})}" class="btn btn-edit">Edit
 <a th:href="@{/user/delete/{id} (id=${user.id})}" class="btn btn-delete" onclick="return confirm('Are you sure you want to delete this user?')">Delete
 </td>
 </tr>
 <tr th:if="${users.isEmpty()}">
 <td colspan="3">No users found</td>
 </tr>
 </tbody>
 </table>
</div>
</body>
</html>
```



- **paginated-user-list.html:** Displays a paginated user list with controls to select page size (e.g., 5, 10, 20) and navigate pages (Previous, Next, page numbers). Uses data from PaginatedUserService.getPaginatedUsers.
- ```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Paginated User List</title>
    <meta charset="UTF-8"/>
    <link rel="stylesheet" th:href="@{/resources/css/styles.css}"/>
</head>
<body>
<div class="container">
    <h1>Paginated User List</h1>
    <div th:if="${errorMessage}" class="error-message" th:text="${errorMessage}"></div>
    <a th:href="@{/user/add}" class="btn btn-add">Add New User</a>
    <a th:href="@{}" class="btn btn-nav">Back to Default List</a>
    <a th:href="@{/sorted-user-list}" class="btn btn-nav">View Sorted List</a>

    <!-- Pagination controls -->
    <div class="pagination">
        <form th:action="@{/paginated-user-list}" method="get">
            <label for="size">Users per page:</label>
            <select id="size" name="size" onchange="this.form.submit()">
                <option th:value="5" th:selected="${pageSize == 5}">5</option>
                <option th:value="10" th:selected="${pageSize == 10}">10</option>
                <option th:value="20" th:selected="${pageSize == 20}">20</option>
            </select>
            <input type="hidden" name="page" th:value="${currentPage}"/>
        </form>
    </div>

    <table>
        <thead>
            <tr>
                <th>ID</th>
                <th>Name</th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody>
            <tr th:each="user : ${users}">
                <td th:text="${user.id}"></td>
                <td th:text="${user.name}"></td>
                <td>
                    <a th:href="@{/user/edit/{id}(id=${user.id})}" class="btn btn-edit">Edit</a>
                    <a th:href="@{/user/delete/{id}(id=${user.id})}" class="btn btn-delete" onclick="return confirm('Are you sure you want to delete this user?')">Delete</a>
                </td>
            </tr>
            <tr th:if="${users.isEmpty()}">

```



```

        <td colspan="3">No users found</td>
    </tr>
</tbody>
</table>

<!-- Pagination navigation --&gt;
&lt;div class="pagination"&gt;
    &lt;th:block th:with="totalPages=${totalUsers / pageSize + (totalUsers % pageSize &gt; 0 ? 1 : 0)}"&gt;
        &lt;a th:if="${currentPage &gt; 1}" th:href="@{/paginated-user-list(page=${currentPage - 1}, size=${pageSize})}" class="btn btn-nav"&gt;Previous&lt;/a&gt;
        &lt;span th:each="i : ${#numbers.sequence(1, totalPages)}"&gt;
            &lt;a th:href="@{/paginated-user-list(page=${i}, size=${pageSize})}" th:class="${i == currentPage} ? 'btn btn-nav active' : 'btn btn-nav'" th:text="${i}"&gt;&lt;/a&gt;
        &lt;/span&gt;
        &lt;a th:if="${currentPage &lt; totalPages}" th:href="@{/paginated-user-list(page=${currentPage + 1}, size=${pageSize})}" class="btn btn-nav"&gt;Next&lt;/a&gt;
    &lt;/th:block&gt;
&lt;/div&gt;
&lt;/body&gt;
&lt;/html&gt;
</pre>

```

- **sorted-user-list.html:** Displays a sorted user list with dropdowns to sort by ID or Name (ascending/descending). Uses data from PaginatedUserService.getSortedUsers.

- <!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <title>Sorted User List</title>
 <meta charset="UTF-8"/>
 <link rel="stylesheet" th:href="@{/resources/css/styles.css}"/>
</head>
<body>
<div class="container">
 <h1>Sorted User List</h1>
 <div th:if="\${errorMessage}" class="error-message" th:text="\${errorMessage}"></div>
 <a th:href="@{/user/add}" class="btn btn-add">Add New User
 <a th:href="@{}" class="btn btn-nav">Back to Default List
 <a th:href="@{/paginated-user-list}" class="btn btn-nav">View Paginated List

 <!-- Sorting controls -->
 <div class="sorting">
 <form th:action="@{/sorted-user-list}" method="get">
 <label for="sortBy">Sort by:</label>
 <select id="sortBy" name="sortBy" onchange="this.form.submit()">
 <option value="id" th:selected="\${sortBy == 'id'}">ID</option>
 <option value="name" th:selected="\${sortBy == 'name'}">Name</option>
 </select>
 </form>
 </div>
</div>
</body>
</html>



```

        'name' } " > Name </option>
    </select>
    <label for="sortDir" > Direction: </label>
    <select id="sortDir" name="sortDir"
onchange="this.form.submit() " >
        <option value="asc" th:selected="${sortDir ==
'asc' } " > Ascending </option>
        <option value="desc" th:selected="${sortDir ==
'desc' } " > Descending </option>
    </select>
</form>
</div>

<table>
    <thead>
        <tr>
            <th> ID </th>
            <th> Name </th>
            <th> Actions </th>
        </tr>
    </thead>
    <tbody>
        <tr th:each="user : ${users}" >
            <td th:text="${user.id}" ></td>
            <td th:text="${user.name}" ></td>
            <td>
                <a th:href="@{/user/edit/{id} (id=${user.id}) } "
class="btn btn-edit" > Edit </a>
                <a th:href="@{/user/delete/{id} (id=${user.id}) } "
class="btn btn-delete"
                    onclick="return confirm('Are you sure you want to
delete this user?') " > Delete </a>
            </td>
        </tr>
        <tr th:if="${users.isEmpty() } " >
            <td colspan="3" > No users found </td>
        </tr>
    </tbody>
</table>
</div>
</body>
</html>

```

- **user-form.html:** Provides a form for creating or editing users, used by /user/add and /user/edit/{id} endpoints. Displays error messages for invalid inputs.
- <!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <title>Add/Edit User</title>
 <meta charset="UTF-8" />
 <link rel="stylesheet" th:href="@{/resources/css/styles.css} "/>
</head>
<body>
<div class="container">
 <h1 th:text="\${user.id} ? 'Edit User' : 'Add User'" ></h1>
 <div th:if="\${errorMessage}" class="error-message"
th:text="\${errorMessage}" ></div>
 <form th:action="\${user.id} ? @{/user/edit/{id} (id=\${user.id}) } :



```

@{/user/add}" th:object="${user}" method="post">
    <label for="id">ID:</label>
    <input type="text" id="id" th:field="*{id}"
th:readonly="${user.id != null}"/>
    <br/>
    <label for="name">Name:</label>
    <input type="text" id="name" th:field="*{name}"/>
    <br/>
    <button type="submit">Save</button>
    <a th:href="@{}" class="cancel">Cancel</a>
</form>
</div>
</body>
</html>

```

- **pom.xml:** Maven configuration file defining project dependencies (Spring 6, Thymeleaf, MySQL Connector, HikariCP, EhCache, Logback, Jakarta Servlet API) and build settings (Java 21, -parameters flag for caching, WAR packaging).

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.example</groupId>
    <artifactId>spring-jdbc-caching</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>spring-jdbc-caching Maven Webapp</name>
    <url>http://maven.apache.org</url>

    <properties>
        <java.version>21</java.version>
        <spring.version>6.1.14</spring.version>
        <thymeleaf.version>3.1.2.RELEASE</thymeleaf.version>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- Spring Core -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <!-- Spring Context Support for JCache -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context-support</artifactId>

```



```
<version>${spring.version}</version>
</dependency>

<!-- Thymeleaf -->
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf-spring6</artifactId>
    <version>${thymeleaf.version}</version>
</dependency>

<!-- MySQL Connector -->
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>9.1.0</version>
</dependency>

<!-- HikariCP -->
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <version>6.0.0</version>
</dependency>

<!-- EhCache -->
<dependency>
    <groupId>org.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>3.10.8</version>
</dependency>

<!-- Servlet API (provided by Tomcat) -->
<dependency>
    <groupId>jakarta.servlet</groupId>
    <artifactId>jakarta.servlet-api</artifactId>
    <version>6.1.0</version>
    <scope>provided</scope>
</dependency>

<!-- Logging -->
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.5.12</version>
</dependency>
</dependencies>

<build>
    <finalName>spring-jdbc-caching</finalName>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>3.4.0</version>
            <configuration>
                <failOnMissingWebXml>false</failOnMissingWebXml>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
```



```
<version>3.13.0</version>
<configuration>
    <source>${java.version}</source>
    <target>${java.version}</target>
    <compilerArgs>
        <arg>-parameters</arg>
    </compilerArgs>
</configuration>
</plugin>
</plugins>
</build>
</project>
```

- **src/test/java/com/example/**: Placeholder for test files (not implemented in this description), which could include unit and integration tests for services and controllers.

Key Features and Implementation Details

- **CRUD Operations**: Managed by UserService and UserController, with endpoints for creating, reading, updating, and deleting users. Thymeleaf templates (index.html, user-form.html) provide the UI.
- **Caching**: EhCache with JCache caches user data in usersCache. UserService caches individual users (getUserById) and the full list (getAllUsers), with @CachePut and @CacheEvict ensuring updates and deletions maintain cache consistency. PaginatedUserService caches paginated and sorted results with unique keys.
- **Pagination**: Implemented server-side in PaginatedUserService.getPaginatedUsers using SQL LIMIT and OFFSET. paginated-user-list.html renders page navigation and size selection.
- **Sorting**: Implemented server-side in PaginatedUserService.getSortedUsers using SQL ORDER BY. sorted-user-list.html provides dropdowns for sorting options.
- **Transaction Management**: @Transactional in UserService ensures database consistency for create, update, and delete operations. CacheConfig enables transaction-aware caching to synchronize cache updates with transaction commits.
- **UI Navigation**: index.html includes links to paginated-user-list.html and sorted-user-list.html. The new templates link back to index.html and each other, ensuring seamless navigation.
- **Error Handling**: Displays user-friendly error messages in all templates for invalid inputs or failed operations, using RedirectAttributes in controllers.
- **Deployment**: Packaged as a WAR file and deployed on Tomcat 11, accessible at <http://localhost:8080/spring-jdbc-thymeleaf>.

This structure and implementation provide a robust, scalable CRUD application with efficient data management and a user-friendly interface, while isolating pagination and sorting functionality to avoid modifying core components.



User Management

Initially due to `@cacheable`, reads all data from the database, as duration set to 1 Min, next read before one minute give the same data so edit within 1 min not reflected. it will reflect reload after the 1 min

Add New User | View Paginated List | View Sorted List

ID	Name	Actions
1	Alan Pawar	<button>Edit</button> <button>Delete</button>
11	amol Kumar	<button>Edit</button> <button>Delete</button>
111	Mark Kadam	<button>Edit</button> <button>Delete</button>
123	soham shinde	<button>Edit</button> <button>Delete</button>
124	mahadev	<button>Edit</button> <button>Delete</button>
21	kiran More-patil	<button>Edit</button> <button>Delete</button>

Add/Edit User

localhost:8080/spring-jdbc-caching/user/edit/1

Edit User

ID:

Name:

Save Cancel



Paginated User List

ID	Name	Actions
1	Alan Pawar	<button>Edit</button> <button>Delete</button>
11	amol Kumar	<button>Edit</button> <button>Delete</button>
111	Mark Kadam	<button>Edit</button> <button>Delete</button>
123	soham shinde	<button>Edit</button> <button>Delete</button>
124	mahadev	<button>Edit</button> <button>Delete</button>

Users per page: 5

1 2 Next

Sorted User List

ID	Name	Actions
123	soham shinde	<button>Edit</button> <button>Delete</button>
2121	satish chavan	<button>Edit</button> <button>Delete</button>
111	Mark Kadam	<button>Edit</button> <button>Delete</button>
124	mahadev	<button>Edit</button> <button>Delete</button>
21	kiran More-patil	<button>Edit</button> <button>Delete</button>
11	amol Kumar	<button>Edit</button> <button>Delete</button>

Sort by: Name Direction: Descending

Add New User Back to Default List View Paginated List



What is ORM?

Object-Relational Mapping (ORM) is a programming technique that allows developers to map object-oriented programming constructs (like Java classes and objects) to relational database tables. It bridges the gap between the object-oriented world of application code and the relational world of databases. ORM enables developers to work with database entities as objects in their programming language, abstracting away much of the SQL and database-specific details.

Key features of ORM include:

- **Mapping:** Defining relationships between Java classes and database tables, and between class fields and table columns.
- **CRUD Operations:** Performing Create, Read, Update, and Delete operations on database records using object-oriented syntax.
- **Querying:** Using object-oriented query languages or APIs instead of raw SQL.
- **Transaction Management:** Handling database transactions within the application code.
- **Data Consistency:** Ensuring that the object state and database state remain synchronized.

Popular ORM frameworks in Java include Hibernate, EclipseLink, and OpenJPA.

ORM in the Spring Framework (Spring Core, Not Spring Boot)

In the context of the **Spring Framework** (specifically **Spring Core** and related modules like **Spring ORM**), ORM is achieved by integrating with an ORM framework (e.g., Hibernate) and leveraging Spring's infrastructure for dependency injection, transaction management, and configuration. Spring Core itself does not provide an ORM implementation but acts as a facilitator to integrate and simplify the use of ORM frameworks like Hibernate.

Here's how ORM is achieved in the Spring Framework (focusing on Spring Core, not Spring Boot):

1. Integration with ORM Frameworks

Spring Core provides the **Spring ORM** module, which offers seamless integration with popular ORM frameworks like Hibernate, JPA (Java Persistence API), and others. It simplifies configuration, transaction management, and exception handling.

- **Key Components:**
 - **Hibernate/JPA Support:** Spring provides templates (e.g., `HibernateTemplate`) and support classes (e.g., `LocalSessionFactoryBean` for Hibernate or `LocalContainerEntityManagerFactoryBean` for JPA) to configure and manage ORM sessions.
 - **Dependency Injection:** Spring's Inversion of Control (IoC) container is used to inject ORM-related beans (e.g., `SessionFactory` for Hibernate or `EntityManagerFactory` for JPA) into application components.
 - **Consistent Exception Handling:** Spring translates ORM-specific exceptions (e.g., Hibernate's `HibernateException`) into its own `DataAccessException` hierarchy, making exception handling consistent across different ORM frameworks.



2. Configuration of ORM in Spring Core

To use an ORM framework like Hibernate with Spring Core, you typically configure the necessary beans in a Spring configuration file (e.g., XML-based or Java-based configuration). Here's an example using XML configuration for Hibernate:

```
<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>

    <!-- Hibernate SessionFactory -->
    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="packagesToScan" value="com.example.model"/>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
                <prop key="hibernate.show_sql">true</prop>
                <prop key="hibernate.hbm2ddl.auto">update</prop>
            </props>
        </property>
    </bean>

    <!-- Hibernate Template (Optional) -->
    <bean id="hibernateTemplate" class="org.springframework.orm.hibernate5.HibernateTemplate">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <!-- Transaction Manager -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate5.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

</beans>
```



- **DataSource:** Configures the database connection (e.g., using DriverManagerDataSource for simplicity or a connection pool like Apache DBCP).
- **SessionFactory:** The LocalSessionFactoryBean creates a Hibernate SessionFactory, which is the core component for interacting with the database in Hibernate.
- **Hibernate Properties:** Configures Hibernate settings like the database dialect, SQL logging, and schema generation.
- **Transaction Manager:** The HibernateTransactionManager integrates Hibernate transactions with Spring's transaction management.

3. Using ORM in Application Code

Once configured, you can inject the SessionFactory or HibernateTemplate into your DAOs (Data Access Objects) or services to perform database operations.

Example DAO using HibernateTemplate:

```
import org.springframework.orm.hibernate5.HibernateTemplate;

public class UserDao {
    private HibernateTemplate hibernateTemplate;

    public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
        this.hibernateTemplate = hibernateTemplate;
    }

    public void saveUser(User user) {
        hibernateTemplate.save(user);
    }

    public User getUser(Long id) {
        return hibernateTemplate.get(User.class, id);
    }
}
```

Alternatively, you can work directly with the SessionFactory for more control:

```
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
public class UserDao {
    @Autowired
    private SessionFactory sessionFactory;

    public void saveUser(User user) {
        sessionFactory.getCurrentSession().save(user);
    }
}
```



```
public User getUser(Long id) {  
    return sessionFactory.getCurrentSession().get(User.class, id);  
}  
}
```

4. Transaction Management

Spring Core provides robust transaction management for ORM operations. You can use declarative transaction management with annotations or XML.

- **Using Annotations:** Enable annotation-based transaction management in the configuration:

```
<bean id="transactionManager"  
      class="org.springframework.orm.hibernate5.HibernateTransactionManager">  
    <property name="sessionFactory" ref="sessionFactory"/>  
</bean>
```

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

Then, use the `@Transactional` annotation in your service layer:

```
java  
CollapseWrap  
Copy  
import org.springframework.transaction.annotation.Transactional;
```

```
@Transactional  
public class UserService {  
    @Autowired  
    private UserDao userDao;  
  
    public void saveUser(User user) {  
        userDao.saveUser(user);  
    }  
}
```

- **Using XML:** Define transaction advice and apply it to methods in XML configuration.

5. Benefits of Using Spring Core with ORM

- **Simplified Configuration:** Spring abstracts much of the boilerplate code required to set up an ORM framework.
- **Transaction Management:** Spring's declarative transaction management simplifies handling transactions across multiple operations.
- **Exception Handling:** Spring's `DataAccessException` hierarchy provides consistent exception handling.
- **Dependency Injection:** Spring's IoC container makes it easy to wire ORM components into your application.



- **Integration with Other Spring Modules:** Spring ORM integrates seamlessly with Spring's JDBC, AOP, and other modules.

6. Key Classes in Spring ORM (Spring Core)

- LocalSessionFactoryBean: Configures a Hibernate SessionFactory.
- HibernateTemplate: Simplifies Hibernate operations (though less commonly used in modern Spring applications).
- HibernateTransactionManager: Manages Hibernate transactions.
- LocalContainerEntityManagerFactoryBean: Configures a JPA EntityManagerFactory.
- JpaTransactionManager: Manages JPA transactions.

7. Example Entity

Here's an example of a simple Hibernate entity:

```
import javax.persistence.Entity;
import javax.persistence.Id;
```

```
@Entity
public class User {
    @Id
    private Long id;
    private String name;
    private String email;

    // Getters and setters
    public Long getId() { return id; }
    public void setId(Long 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; }
}
```

This entity maps to a User table in the database, with columns id, name, and email.

Key Differences from Spring Boot

Since the question specifies Spring Core (not Spring Boot), note that Spring Boot simplifies ORM configuration further with auto-configuration and starter dependencies (e.g., spring-boot-starter-data-jpa). In Spring Core, you manually configure the DataSource, SessionFactory, and transaction management, as shown above, without relying on Spring Boot's conventions or auto-configuration.

ORM in the Spring Framework (Spring Core) is achieved by integrating with an ORM framework like Hibernate or JPA, using Spring's ORM module. Spring provides configuration support, dependency injection, transaction management, and exception handling to simplify ORM usage. You configure the SessionFactory or EntityManagerFactory, use Spring's



transaction management, and perform database operations via DAOs or services, leveraging Spring's IoC container for wiring components. This approach offers flexibility and control, though it requires more manual configuration compared to Spring Boot.

Spring ORM Syllabus (Using Spring Core)

Chapter 1: Introduction to ORM and Spring ORM

- **1.1 Overview of Object-Relational Mapping (ORM)**
 - Definition and purpose of ORM
 - Challenges of object-relational impedance mismatch
 - Benefits of ORM in Java applications
- **1.2 Role of Spring ORM in Spring Core**
 - Integration with ORM frameworks (Hibernate, JPA, etc.)
 - Spring's ORM philosophy: abstraction and simplification
 - Comparison with direct ORM framework usage
- **1.3 Spring ORM vs. Spring Boot Auto-Configuration**
 - Manual configuration in Spring Core
 - Key differences from Spring Boot's ORM setup
- **1.4 Supported ORM Frameworks**
 - Hibernate
 - Java Persistence API (JPA)
 - Other frameworks (e.g., TopLink, OpenJPA)

Chapter 2: Setting Up Spring ORM Environment

- **2.1 Prerequisites**
 - Java Development Kit (JDK) setup
 - Maven/Gradle for dependency management
 - Database setup (e.g., MySQL, PostgreSQL)
- **2.2 Spring Core Dependencies**
 - Adding Spring Core and Spring ORM dependencies
 - Including Hibernate/JPA dependencies
- **2.3 Configuring the Spring Application Context**
 - XML-based configuration
 - Java-based configuration
 - Annotation-driven configuration
- **2.4 Database Connectivity**
 - Configuring a DataSource (e.g., DriverManagerDataSource, Apache DBCP)
 - Connection pooling basics

Chapter 3: Configuring Hibernate with Spring Core

- **3.1 Introduction to Hibernate**
 - Hibernate architecture and components
 - Hibernate Session and SessionFactory
- **3.2 Configuring Hibernate SessionFactory**
 - Using LocalSessionFactoryBean
 - Setting up data source and Hibernate properties



- Mapping entities (XML vs. annotations)
- **3.3 Hibernate Properties**
 - Database dialect configuration
 - Schema generation (hbm2ddl.auto)
 - SQL logging and formatting
- **3.4 HibernateTemplate**
 - Overview and usage
 - Pros and cons of using HibernateTemplate
 - Direct SessionFactory access as an alternative
- **3.5 Entity Mapping**
 - Creating Hibernate entities with annotations
 - Mapping relationships (one-to-one, one-to-many, many-to-many)
 - Configuring primary keys and composite keys

Chapter 4: Configuring JPA with Spring Core

- **4.1 Introduction to JPA**
 - JPA vs. Hibernate: Key differences
 - JPA providers (Hibernate, EclipseLink, etc.)
- **4.2 Configuring EntityManagerFactory**
 - Using LocalContainerEntityManagerFactoryBean
 - Setting up persistence unit and data source
- **4.3 JPA Annotations**
 - @Entity, @Table, @Id, @Column
 - Mapping relationships (@OneToMany, @ManyToOne, etc.)
 - Inheritance strategies (@Inheritance, @MappedSuperclass)
- **4.4 JPA Configuration**
 - persistence.xml setup
 - JPA properties (e.g., dialect, show_sql)
 - Integrating JPA with Spring's dependency injection

Chapter 5: Data Access with Spring ORM

- **5.1 Data Access Objects (DAOs)**
 - Creating DAOs for Hibernate and JPA
 - Injecting SessionFactory or EntityManager
- **5.2 CRUD Operations**
 - Creating records (save/persist)
 - Reading records (get/find)
 - Updating records (update/merge)
 - Deleting records
- **5.3 Using HibernateTemplate**
 - Simplifying CRUD with HibernateTemplate
 - Limitations and best practices
- **5.4 Direct Session/EntityManager Usage**
 - Working with Session in Hibernate
 - Working with EntityManager in JPA
 - Managing sessions and entity managers



Chapter 6: Transaction Management

- **6.1 Introduction to Transactions in Spring**
 - ACID properties and transaction basics
 - Spring's transaction management abstractions
- **6.2 Configuring Transaction Managers**
 - HibernateTransactionManager for Hibernate
 - JpaTransactionManager for JPA
- **6.3 Declarative Transaction Management**
 - Using @Transactional annotation
 - XML-based transaction configuration
 - Transaction propagation and isolation levels
- **6.4 Programmatic Transaction Management**
 - Using TransactionTemplate
 - Manual transaction control with PlatformTransactionManager
- **6.5 Transaction Best Practices**
 - Managing transaction boundaries
 - Handling transaction rollback
 - Avoiding common pitfalls (e.g., lazy loading issues)

Chapter 7: Querying with Spring ORM

- **7.1 Querying in Hibernate**
 - HQL (Hibernate Query Language)
 - Criteria API
 - Native SQL queries
- **7.2 Querying in JPA**
 - JPQL (Java Persistence Query Language)
 - Criteria API
 - Native queries
- **7.3 Spring Data Access Utilities**
 - Using HibernateTemplate for queries
 - Simplifying queries with Spring's DAO support
- **7.4 Named Queries**
 - Defining and using named queries
 - Configuring named queries in XML and annotations
- **7.5 Handling Query Results**
 - Fetching single entities
 - Fetching collections and projections
 - Pagination and sorting

Chapter 8: Exception Handling

- **8.1 Spring's DataAccessException Hierarchy**
 - Overview of DataAccessException
 - Mapping ORM-specific exceptions to Spring exceptions
- **8.2 Handling Hibernate Exceptions**
 - Common Hibernate exceptions (e.g., ConstraintViolationException)
 - Exception translation in Spring



- **8.3 Handling JPA Exceptions**
 - JPA-specific exceptions (e.g., PersistenceException)
 - Spring's exception handling for JPA
- **8.4 Custom Exception Handling**
 - Defining application-specific exceptions
 - Integrating with Spring's exception handling

Chapter 9: Advanced ORM Features

- **9.1 Lazy vs. Eager Loading**
 - Configuring fetch strategies
 - Handling LazyInitializationException
- **9.2 Caching**
 - First-level cache (session cache)
 - Second-level cache configuration
 - Query cache setup
- **9.3 Batch Processing**
 - Batch inserts and updates
 - Optimizing performance with batch operations
- **9.4 Auditing and Versioning**
 - Using @Version for optimistic locking
 - Implementing audit fields (e.g., created, updated timestamps)
- **9.5 Interceptors and Listeners**
 - Hibernate interceptors
 - JPA entity listeners

Chapter 10: Best Practices and Performance Optimization

- **10.1 Configuration Best Practices**
 - Optimizing SessionFactory and EntityManagerFactory
 - Choosing appropriate connection pooling
- **10.2 Performance Tuning**
 - Reducing database round-trips
 - Optimizing fetch strategies
 - Avoiding N+1 select issues
- **10.3 Testing ORM Code**
 - Writing unit tests for DAOs
 - Using in-memory databases (e.g., H2) for testing
- **10.4 Debugging and Logging**
 - Enabling SQL logging
 - Troubleshooting common ORM issues
- **10.5 Migration Considerations**
 - Migrating between ORM frameworks
 - Upgrading Hibernate/JPA versions

Chapter 11: Integration with Other Spring Modules

- **11.1 Spring Core Integration**
 - Using Spring's IoC container for dependency injection
 - Configuring beans for ORM components



- **11.2 Spring JDBC Integration**
 - Combining Spring JDBC with Spring ORM
 - Using JdbcTemplate alongside ORM
- **11.3 Spring AOP**
 - Applying aspects to ORM operations
 - Implementing cross-cutting concerns (e.g., logging, security)
- **11.4 Spring MVC Integration**
 - Using ORM in web applications
 - Managing transactions in controllers and services

Chapter 12: Case Study and Practical Example

- **12.1 Building a Sample Application**
 - Setting up a Spring Core project with Hibernate
 - Implementing a simple CRUD application
- **12.2 Configuring JPA in a Real-World Scenario**
 - Setting up a JPA-based application
 - Handling complex relationships
- **12.3 Transaction Management in Practice**
 - Implementing a multi-step transaction
 - Testing transaction rollback
- **12.4 Performance Optimization Example**
 - Optimizing a query-heavy application
 - Applying caching and batch processing

Introduction to JPA with Spring Core

Java Persistence API (JPA) is a Java specification for Object-Relational Mapping (ORM) that provides a standardized way to map Java objects to relational database tables. It abstracts the complexities of database interactions, allowing developers to work with entities (Java objects) instead of raw SQL. JPA is not an implementation but a specification, with providers like Hibernate, EclipseLink, and OpenJPA offering concrete implementations.

Spring Core, the foundational module of the Spring Framework, integrates seamlessly with JPA to simplify its configuration, usage, and management in enterprise applications. Unlike Spring Boot, which relies on auto-configuration, Spring Core requires manual setup, giving developers fine-grained control over JPA configuration, transaction management, and dependency injection. This introduction focuses on using JPA with Spring Core (not Spring Boot), leveraging Spring's Inversion of Control (IoC) container and ORM module to integrate JPA effectively.

Key Concepts of JPA with Spring Core

1. JPA Basics:

- **Entities:** Java classes annotated with @Entity that map to database tables.
- **EntityManager:** The primary interface for interacting with the persistence context, handling CRUD operations, queries, and transactions.



- **EntityManagerFactory:** Creates EntityManager instances and manages the persistence unit.
- **Persistence Unit:** A logical grouping of entities defined in a persistence.xml file or programmatically.
- **Annotations:** JPA uses annotations like @Id, @Table, @Column, @OneToMany, etc., to define mappings.

2. Spring Core's Role:

- **Dependency Injection:** Spring's IoC container manages JPA components like EntityManagerFactory and DAOs.
- **Transaction Management:** Spring provides declarative and programmatic transaction management for JPA operations.
- **Exception Handling:** Spring translates JPA-specific exceptions into its DataAccessException hierarchy.
- **Configuration:** Spring Core allows manual configuration of JPA via XML or Java-based configuration, offering flexibility for complex setups.

Setting Up JPA with Spring Core

To integrate JPA with Spring Core, you configure the necessary components (e.g., EntityManagerFactory, DataSource, and transaction manager) using Spring's configuration mechanisms. Below is a step-by-step guide to setting up JPA with Hibernate as the JPA provider.

1. Dependencies

Add the required dependencies to your project (e.g., using Maven):

```
<dependencies>
    <!-- Spring Core -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.34</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>5.3.34</version>
    </dependency>
    <!-- Hibernate as JPA Provider -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.6.15.Final</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
```



```
<version>5.6.15.Final</version>
</dependency>
<!-- Database Driver (e.g., MySQL) -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
</dependency>
</dependencies>
```

2. Configuring the DataSource

Define a DataSource to connect to the database. For simplicity, use DriverManagerDataSource (in production, consider a connection pool like HikariCP).

```
<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>
```

3. Configuring EntityManagerFactory

Use LocalContainerEntityManagerFactoryBean to configure the JPA EntityManagerFactory.

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="packagesToScan" value="com.example.model"/>
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"/>
    </property>
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
        </props>
    </property>
</bean>
```

- **packagesToScan:** Specifies the package containing JPA entities.
- **jpaVendorAdapter:** Configures Hibernate as the JPA provider.
- **jpaProperties:** Sets provider-specific properties (e.g., Hibernate settings).

Alternatively, use Java-based configuration:

```
@Configuration
```



```

public class JpaConfig {
    @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 LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean emf = new
        LocalContainerEntityManagerFactoryBean();
        emf.setDataSource(dataSource());
        emf.setPackagesToScan("com.example.model");
        emf.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
        Properties jpaProperties = new Properties();
        jpaProperties.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");
        jpaProperties.setProperty("hibernate.show_sql", "true");
        jpaProperties.setProperty("hibernate.hbm2ddl.auto", "update");
        emf.setJpaProperties(jpaProperties);
        return emf;
    }
}

```

4. Configuring Transaction Management

Use JpaTransactionManager to manage JPA transactions.

```

<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>

```

In Java-based configuration:

```

@Bean
public JpaTransactionManager transactionManager(EntityManagerFactory emf) {
    JpaTransactionManager transactionManager = new JpaTransactionManager();
    transactionManager.setEntityManagerFactory(emf);
    return transactionManager;
}

```



```
}
```

Enable @Transactional annotations:

```
@Configuration  
@EnableTransactionManagement  
public class JpaConfig {  
    // Other beans as above  
}
```

5. Creating a JPA Entity

Define a simple entity class:

```
import javax.persistence.Entity;  
import javax.persistence.Id;  
  
@Entity  
public class User {  
    @Id  
    private Long id;  
    private String name;  
    private String email;  
  
    // Getters and setters  
    public Long getId() { return id; }  
    public void setId(Long 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; }  
}
```

6. Implementing a DAO

Create a Data Access Object (DAO) to interact with the database using EntityManager.

```
import javax.persistence.EntityManager;  
import javax.persistence.PersistenceContext;  
import org.springframework.stereotype.Repository;  
  
@Repository  
public class UserDao {  
    @PersistenceContext  
    private EntityManager entityManager;  
  
    public void saveUser(User user) {
```



```
        entityManager.persist(user);
    }

    public User getUser(Long id) {
        return entityManager.find(User.class, id);
    }
}
```

- **@PersistenceContext**: Injects the EntityManager managed by Spring.
- **@Repository**: Marks the DAO as a Spring-managed bean and enables exception translation.

7. Using Transactions

Use @Transactional in the service layer to manage transactions.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.transaction.annotation.Transactional;
```

```
@Service
public class UserService {
    @Autowired
    private UserDao userDao;

    @Transactional
    public void saveUser(User user) {
        userDao.saveUser(user);
    }

    @Transactional(readOnly = true)
    public User getUser(Long id) {
        return userDao.getUser(id);
    }
}
```

Key Features of JPA with Spring Core

1. Dependency Injection:

- Spring's IoC container manages EntityManagerFactory, DataSource, and DAOs, reducing boilerplate code.
- @PersistenceContext injects a thread-safe EntityManager.

2. Transaction Management:

- Declarative transactions with @Transactional simplify transaction handling.
- JpaTransactionManager integrates JPA transactions with Spring's transaction infrastructure.

3. Exception Handling:



- Spring translates JPA's PersistenceException into DataAccessException subclasses, providing consistent error handling.
- Example: EntityNotFoundException becomes EmptyResultDataAccessException.

4. Querying:

- Use JPQL or Criteria API for queries via EntityManager.
- Example JPQL query:

```
public List<User> findUsersByName(String name) {
    return entityManager.createQuery("SELECT u FROM User u WHERE u.name = :name",
        User.class)
        .setParameter("name", name)
        .getResultList();
}
```

5. Flexibility:

- Spring Core allows full control over JPA configuration, unlike Spring Boot's auto-configuration.
- Customize JPA provider settings, connection pools, and transaction strategies.

Benefits of Using JPA with Spring Core

- **Simplified Configuration:** Spring abstracts JPA setup (e.g., EntityManagerFactory) via beans.
- **Transaction Integration:** Seamless transaction management across JPA and other Spring components.
- **Exception Translation:** Consistent exception handling across JPA providers.
- **Modularity:** Integrates with other Spring modules (e.g., Spring MVC, Spring JDBC).
- **Flexibility:** Manual configuration suits complex or custom requirements.

The provided code is a simple console-based Java application that demonstrates CRUD (Create, Read, Update, Delete) operations using JPA (Java Persistence API) with Hibernate as the implementation and MySQL as the database. Below is a description of the application:

Overview

The application manages a User entity with attributes for id, name, and email, stored in a MySQL database named jpa_crud_db. It uses a Maven-based project structure and provides a menu-driven console interface to perform CRUD operations on the User entity.

Components

1. Project Configuration (pom.xml)

- A Maven configuration file that includes dependencies for Hibernate (hibernate-core) and MySQL Connector (mysql-connector-java).
- Specifies Java 11 as the source and target version.

```
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```



```

<groupId>org.example</groupId>
<artifactId>simple-jpa-mysql-crud</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>

<name>simple-jpa-mysql-crud</name>
<url>http://maven.apache.org</url>

<properties>
    <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
</properties>

<dependencies>
    <!-- JPA/Hibernate -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.6.15.Final</version>
    </dependency>
    <!-- MySQL Connector -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.33</version>
    </dependency>
</dependencies>
</project>

```

3. JPA Configuration (`persistence.xml`)

- Located in `src/main/resources/META-INF/`.
- Configures the persistence unit (`myPU`) for Hibernate with MySQL.
- Defines database connection properties (URL, user, password, driver) and Hibernate settings (e.g., `hibernate.hbm2ddl.auto=update` to automatically create/update the database schema, and `hibernate.show_sql=true` to log SQL statements).

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
    <persistence-unit name="myPU" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <class>com.example.entity.User</class>
        <properties>
            <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/userdb"/>
            <property name="javax.persistence.jdbc.user" value="root"/>
            <property name="javax.persistence.jdbc.password" value="Archer@12345"/>
            <property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver"/>
            <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL8Dialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>

```



4. Entity Class (User.java)

- A JPA entity class representing a user in the database.
- Contains fields: id (auto-incremented primary key), name, and email.
- Includes annotations like @Entity, @Id, and @GeneratedValue for JPA mapping.
- Provides constructors, getters, setters, and a `toString` method for easy representation.

```
5. package com.example.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;

    // Constructors
    public User() {}

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and Setters
    public Long getId() { return id; }
    public void setId(Long 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 "User{id=" + id + ", name='" + name + "', email='" +
email + "'}";
    }
}
```

6. Data Access Object (UserDAO.java)

- Handles database operations using JPA's EntityManager.
- Provides methods for:
 - Creating a new user (`createUser`)
 - Retrieving a user by ID (`getUser`)
 - Listing all users (`getAllUsers`)
 - Updating an existing user (`updateUser`)
 - Deleting a user by ID (`deleteUser`)
- Manages the EntityManagerFactory and ensures proper resource cleanup.



```
7. package com.example.dao;

import com.example.entity.User;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import java.util.List;

public class UserDAO {
    private EntityManagerFactory emf;

    public UserDAO() {
        emf = Persistence.createEntityManagerFactory("myPU");
    }

    public void createUser(User user) {
        EntityManager em = emf.createEntityManager();
        try {
            em.getTransaction().begin();
            em.persist(user);
            em.getTransaction().commit();
        } finally {
            em.close();
        }
    }

    public User getUser(Long id) {
        EntityManager em = emf.createEntityManager();
        try {
            return em.find(User.class, id);
        } finally {
            em.close();
        }
    }

    public List<User> getAllUsers() {
        EntityManager em = emf.createEntityManager();
        try {
            return em.createQuery("SELECT u FROM User u",
User.class).getResultList();
        } finally {
            em.close();
        }
    }

    public void updateUser(User user) {
        EntityManager em = emf.createEntityManager();
        try {
            em.getTransaction().begin();
            em.merge(user);
            em.getTransaction().commit();
        } finally {
            em.close();
        }
    }

    public void deleteUser(Long id) {
        EntityManager em = emf.createEntityManager();
        try {
            em.getTransaction().begin();
            User user = em.find(User.class, id);
            em.remove(user);
            em.getTransaction().commit();
        } finally {
            em.close();
        }
    }
}
```



```
        if (user != null) {
            em.remove(user);
        }
        em.getTransaction().commit();
    } finally {
        em.close();
    }
}

public void close() {
    emf.close();
}
}
```

8. Main Application (Main.java)

- Implements a console-based interface using Scanner for user input.
- Presents a menu with six options:
 1. Create a new user (prompts for name and email).
 2. Read a user by ID.
 3. Update an existing user's name and email.
 4. Delete a user by ID.
 5. List all users.
 6. Exit the application.
 - Interacts with UserDAO to perform the requested CRUD operations and displays results.

```
package com.example;

import com.example.dao.UserDAO;
import com.example.entity.User;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        UserDAO userDAO = new UserDAO();
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("\n== User Management System ==");
            System.out.println("1. Create User");
            System.out.println("2. Read User");
            System.out.println("3. Update User");
            System.out.println("4. Delete User");
            System.out.println("5. List All Users");
            System.out.println("6. Exit");
            System.out.print("Choose an option: ");
        }
    }
}
```



```
int choice = scanner.nextInt();
scanner.nextLine(); // Consume newline

switch (choice) {
    case 1:
        System.out.print("Enter name: ");
        String name = scanner.nextLine();
        System.out.print("Enter email: ");
        String email = scanner.nextLine();
        userDAO.createUser(new User(name, email));
        System.out.println("User created successfully!");
        break;

    case 2:
        System.out.print("Enter user ID: ");
        Long id = scanner.nextLong();
        User user = userDAO.getUser(id);
        System.out.println(user != null ? user : "User not found!");
        break;

    case 3:
        System.out.print("Enter user ID to update: ");
        Long updateId = scanner.nextLong();
        scanner.nextLine();
        User updateUser = userDAO.getUser(updateId);
        if (updateUser != null) {
            System.out.print("Enter new name: ");
            updateUser.setName(scanner.nextLine());
            System.out.print("Enter new email: ");
            updateUser.setEmail(scanner.nextLine());
            userDAO.updateUser(updateUser);
            System.out.println("User updated successfully!");
        } else {
            System.out.println("User not found!");
        }
        break;

    case 4:
        System.out.print("Enter user ID to delete: ");
        Long deleteId = scanner.nextLong();
        userDAO.deleteUser(deleteId);
        System.out.println("User deleted successfully!");
        break;
}
```



```
case 5:  
    System.out.println("All Users:");  
    userDao.getAllUsers().forEach(System.out::println);  
    break;  
  
case 6:  
    userDao.close();  
    scanner.close();  
    System.out.println("Exiting...");  
    return;  
  
default:  
    System.out.println("Invalid option!");  
}  
}  
}  
}
```

Setup Instructions

- **Database:** Create a MySQL database named `jpa_crud_db`.
 - **Configuration:** Update `persistence.xml` with your MySQL credentials (username and password).
 - **Build:** Use Maven (`mvn clean install`) to resolve dependencies and build the project.
 - **Run:** Execute the Main class to start the console application.
 - **Directory Structure:**

```
text
CollapseWrap
Copy
jpa-mysql-crud/
+-- src/
|   +-- main/
|   |   +-- java/
|   |   |   +-- com/example/
|   |   |   |   +-- entity/
|   |   |   |   |   +-- User.java
|   |   |   |   +-- dao/
|   |   |   |   |   +-- UserDAO.java
|   |   +-- resources/
|   |       +-- META-INF/
|   |           +-- persistence.xml
pom.xml
```

Functionality

- **Create:** Users can input a name and email to create a new user record in the database.
 - **Read:** Retrieve and display a user's details by entering their ID.
 - **Update:** Modify a user's name and email by providing their ID and new details.



- **Delete:** Remove a user from the database using their ID.
- **List All:** Display all user records stored in the database.
- **Exit:** Closes the application and releases database resources.

Key Features

- **Simplicity:** The application uses a straightforward console interface, making it easy to understand and use.
- **JPA/Hibernate:** Leverages Hibernate for ORM, abstracting database operations and automatically managing the schema.
- **MySQL Integration:** Connects to a MySQL database for persistent storage.
- **Error Handling:** Basic checks ensure users are notified if a requested user ID does not exist.
- **Resource Management:** Properly closes EntityManager and EntityManagerFactory to prevent resource leaks.

I'll rewrite the JPA MySQL console CRUD application using Spring Core (without Spring Boot) and Java 21, with Java-based Spring configuration instead of XML. The application will maintain the same functionality: a console-based CRUD interface for managing a User entity with id, name, and email fields in a MySQL database. The project will use Maven, Hibernate as the JPA provider, and a Java-based Spring configuration.

Below are the updated artifacts:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>spring-jpa-crud</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- Spring Core -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>6.1.14</version>
        </dependency>
        <dependency>
```



```
<groupId>org.springframework</groupId>
<artifactId>spring-orm</artifactId>
<version>6.1.14</version>
</dependency>
<!-- JPA/Hibernate -->
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-core</artifactId>
<version>6.6.1.Final</version>
</dependency>
<!-- MySQL Connector -->
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>8.0.33</version>
</dependency>
</dependencies>
</project>
```

```
package com.example.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;

    // Constructors
    public User() {}

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and Setters
}
```



```
public Long getId() { return id; }
public void setId(Long 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 "User{id=" + id + ", name='" + name + "', email='" + email + "'}";
}
}
```

```
package com.example.dao;

import com.example.entity.User;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Repository
public class UserDAO {

    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    public void createUser(User user) {
        entityManager.persist(user);
    }

    public User getUser(Long id) {
        return entityManager.find(User.class, id);
    }

    public List<User> getAllUsers() {
        return entityManager.createQuery("SELECT u FROM User u", User.class).getResultList();
    }

    @Transactional
```



```
public void updateUser(User user) {
    entityManager.merge(user);
}

@Transactional
public void deleteUser(Long id) {
    User user = entityManager.find(User.class, id);
    if (user != null) {
        entityManager.remove(user);
    }
}
```

```
package com.example.config;
```

```
import com.example.dao.UserDAO;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
```

```
import javax.sql.DataSource;
import java.util.Properties;
```

```
@Configuration
public class AppConfig {
```

```
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/jpa_crud_db");
        dataSource.setUsername("root");
        dataSource.setPassword("your_password");
        return dataSource;
    }
```

```
    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
```



```
LocalContainerEntityManagerFactoryBean emf = new
LocalContainerEntityManagerFactoryBean();
emf.setDataSource(dataSource());
emf.setPackagesToScan("com.example.entity");
emf.setJpaVendorAdapter(new HibernateJpaVendorAdapter());

Properties jpaProperties = new Properties();
jpaProperties.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQL8Dialect");
jpaProperties.setProperty("hibernate.hbm2ddl.auto", "update");
jpaProperties.setProperty("hibernate.show_sql", "true");
emf.setJpaProperties(jpaProperties);

return emf;
}

@Bean
public JpaTransactionManager transactionManager() {
    JpaTransactionManager transactionManager = new JpaTransactionManager();
    transactionManager.setEntityManagerFactory(entityManagerFactory().getObject());
    return transactionManager;
}

@Bean
public UserDAO userDAO() {
    return new UserDAO();
}
}
```

```
package com.example;

import com.example.config.AppConfig; // Import AppConfig
import com.example.dao.UserDAO;
import com.example.entity.User;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        // Initialize Spring context with Java-based configuration
        try (AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class)) {
```



```
UserDAO userDAO = context.getBean(UserDAO.class);
Scanner scanner = new Scanner(System.in);

while (true) {
    System.out.println("\n==== User Management System ====");
    System.out.println("1. Create User");
    System.out.println("2. Read User");
    System.out.println("3. Update User");
    System.out.println("4. Delete User");
    System.out.println("5. List All Users");
    System.out.println("6. Exit");
    System.out.print("Choose an option: ");

    int choice = scanner.nextInt();
    scanner.nextLine() // Consume newline

    switch (choice) {
        case 1:
            System.out.print("Enter name: ");
            String name = scanner.nextLine();
            System.out.print("Enter email: ");
            String email = scanner.nextLine();
            userDAO.createUser(new User(name, email));
            System.out.println("User created successfully!");
            break;

        case 2:
            System.out.print("Enter user ID: ");
            Long id = scanner.nextLong();
            User user = userDAO.getUser(id);
            System.out.println(user != null ? user : "User not found!");
            break;

        case 3:
            System.out.print("Enter user ID to update: ");
            Long updateId = scanner.nextLong();
            scanner.nextLine();
            User updateUser = userDAO.getUser(updateId);
            if (updateUser != null) {
                System.out.print("Enter new name: ");
                updateUser.setName(scanner.nextLine());
                System.out.print("Enter new email: ");
                updateUser.setEmail(scanner.nextLine());
                userDAO.updateUser(updateUser);
            }
    }
}
```



```
        System.out.println("User updated successfully!");
    } else {
        System.out.println("User not found!");
    }
break;

case 4:
    System.out.print("Enter user ID to delete: ");
    Long deletId = scanner.nextLong();
    userDAO.deleteUser(deletId);
    System.out.println("User deleted successfully!");
    break;

case 5:
    System.out.println("All Users:");
    userDAO.getAllUsers().forEach(System.out::println);
    break;

case 6:
    scanner.close();
    System.out.println("Exiting...");
    return;

default:
    System.out.println("Invalid option!");
}
}
}
}
}
```

Changes Made

1. **Java Version:** Updated to Java 21 in pom.xml (maven.compiler.source and maven.compiler.target).
2. **Dependencies:**
 - Added Spring Core dependencies (spring-context and spring-orm, version 6.1.14).
 - Updated Hibernate to version 6.6.1.Final for compatibility with Java 21 and Spring 6.
 - Kept MySQL Connector (version 8.0.33).
3. **JPA Configuration:**
 - Removed persistence.xml and replaced it with a Java-based Spring configuration in AppConfig.java.
 - AppConfig.java defines beans for:



- DataSource (MySQL connection settings).
- LocalContainerEntityManagerFactoryBean (JPA entity manager factory with Hibernate).
- JpaTransactionManager (transaction management).
- UserDAO (data access object).

4. Entity Class (User.java):

- Updated to use jakarta.persistence imports (JPA 3.0+ for Java 21 compatibility).
- Otherwise unchanged, as the entity structure remains the same.

5. DAO Class (UserDAO.java):

- Replaced manual EntityManagerFactory management with Spring's @PersistenceContext for injecting EntityManager.
- Added @Repository to mark it as a Spring-managed component.
- Used @Transactional for methods that modify the database (createUser, updateUser, deleteUser) to handle transactions via Spring.
- Simplified by removing manual resource cleanup, as Spring manages the EntityManager.

6. Main Class (Main.java):

- Replaced manual EntityManagerFactory initialization with Spring's AnnotationConfigApplicationContext to load the Java-based configuration (AppConfig).
- Retrieved UserDAO as a Spring bean.
- Wrapped the context in a try-with-resources block to ensure proper cleanup.
- Console interface and functionality remain identical to the original.

Project structure:

```
spring-jpa-crud/
|__ src/
|   |__ main/
|   |   |__ java/
|   |   |       |__ com/example/
|   |   |           |__ config/
|   |   |               |__ AppConfig.java
|   |   |           |__ entity/
|   |   |               |__ User.java
|   |   |           |__ dao/
|   |   |               |__ UserDAO.java
|   |   |__ Main.java
|__ pom.xml
```

Integration of Spring-jpa with spring-mvc:

Integrating Spring JPA with Spring MVC in the context of Spring Core (without Spring Boot) involves configuring the Spring framework manually to combine Spring's data access capabilities (via JPA) with its web layer (Spring MVC). This requires setting up the necessary beans, configurations, and dependencies in XML configuration files or Java-based



configuration, as Spring Core does not provide the auto-configuration features of Spring Boot.

Integrating Spring JPA with Spring MVC using Java-based configuration (without Spring Boot) involves setting up Spring Core's dependency injection, configuring Spring MVC for web handling, and integrating Spring Data JPA for database operations. Below, I'll provide a step-by-step explanation of how to achieve this using Java-based configuration, focusing on Spring Core principles like the ApplicationContext and bean management.

Overview

- **Spring Core:** Provides the Inversion of Control (IoC) container via ApplicationContext to manage beans.
- **Spring MVC:** Handles HTTP requests and responses, configured with a DispatcherServlet.
- **Spring JPA:** Uses Spring Data JPA to simplify database interactions with JPA (Java Persistence API).

We'll create a configuration that:

1. Sets up the Spring MVC DispatcherServlet for handling web requests.
2. Configures Spring Data JPA for database access (e.g., using Hibernate as the JPA provider).
3. Integrates both in a Spring Core ApplicationContext using Java-based configuration.

Prerequisites

- Java 8 or higher.
- Maven or Gradle for dependency management.
- A database (e.g., MySQL, PostgreSQL, or H2 for testing).
- Dependencies (add to pom.xml for Maven):

```
<dependencies>
    <!-- Spring Core -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>6.1.14</version>
    </dependency>
    <!-- Spring MVC -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>6.1.14</version>
    </dependency>
    <!-- Spring Data JPA -->
    <dependency>
        <groupId>org.springframework.data</groupId>
```



```
<artifactId>spring-data-jpa</artifactId>
  <version>3.3.5</version>
</dependency>
<!-- Hibernate as JPA provider -->
<dependency>
  <groupId>org.hibernate.orm</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>6.6.1.Final</version>
</dependency>
<!-- Database driver (e.g., H2 for simplicity) -->
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>2.3.230</version>
</dependency>
<!-- Servlet API (for web application) -->
<dependency>
  <groupId>jakarta.servlet</groupId>
  <artifactId>jakarta.servlet-api</artifactId>
  <version>6.1.0</version>
  <scope>provided</scope>
</dependency>
</dependencies>
```

Step-by-Step Integration

1. Project Structure

Assume a basic structure:

```
src/main/java
  |-- com.example.config
    |-- WebConfig.java      // Spring MVC configuration
    |-- JpaConfig.java      // Spring JPA configuration
    |-- AppConfig.java      // Root application context configuration
  |-- com.example.controller
    |-- UserController.java // Spring MVC controller
  |-- com.example.entity
    |-- User.java            // JPA entity
  |-- com.example.repository
    |-- UserRepository.java // Spring Data JPA repository
  |-- com.example.service
    |-- UserService.java    // Service layer
src/main/webapp
  |-- WEB-INF
    |-- web.xml              // Servlet configuration
```

2. Configure the Web Application (web.xml)

Define the DispatcherServlet and root ApplicationContext in WEB-INF/web.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
```



```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
version="6.0">

<!-- Root Application Context -->
<context-param>
    <param-name>contextClass</param-name>
    <param-
value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-value>
</context-param>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>com.example.config.AppConfig</param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<!-- DispatcherServlet for Spring MVC -->
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextClass</param-name>
        <param-
value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-value>
</init-param>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>com.example.config.WebConfig</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

- **Explanation:**

- ContextLoaderListener creates the root ApplicationContext (for services, repositories, and JPA beans) using AppConfig.
- DispatcherServlet creates a child ApplicationContext (for MVC components) using WebConfig.



- Spring Core's AnnotationConfigWebApplicationContext enables Java-based configuration.

Optional Java Based Configuration:

```
package com.example.config;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.ContextLoaderListener;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletRegistration;

public class WebAppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) {
        // Create the root ApplicationContext
        AnnotationConfigWebApplicationContext rootContext = new
AnnotationConfigWebApplicationContext();
        rootContext.register(AppConfig.class); // Register root config (includes JpaConfig)

        // Add ContextLoaderListener to manage the root ApplicationContext
        servletContext.addListener(new ContextLoaderListener(rootContext));

        // Create the DispatcherServlet's ApplicationContext
        AnnotationConfigWebApplicationContext webContext = new
AnnotationConfigWebApplicationContext();
        webContext.register(WebConfig.class); // Register MVC config

        // Configure DispatcherServlet
        DispatcherServlet dispatcherServlet = new DispatcherServlet(webContext);
        ServletRegistration.Dynamic registration = servletContext.addServlet("dispatcher",
dispatcherServlet);
        registration.setLoadOnStartup(1);
        registration.addMapping("/"); // Map DispatcherServlet to root URL
    }
}
```

Explanation of WebAppInitializer.java

- Implements WebApplicationInitializer:
 - This interface is detected by the Servlet 3.0+ container (e.g., Tomcat) at startup, allowing programmatic configuration without web.xml.
 - The onStartup method is called when the application starts.
- Root ApplicationContext:



- An AnnotationConfigWebApplicationContext is created and configured with AppConfig.class, which imports JpaConfig (for JPA setup, including DataSource, EntityManagerFactory, and repositories).
- The ContextLoaderListener is added to the ServletContext to manage the root ApplicationContext, which holds non-web beans like services and repositories.
- MVC ApplicationContext:
 - A separate AnnotationConfigWebApplicationContext is created for the DispatcherServlet and configured with WebConfig.class, which sets up MVC components (e.g., controllers and view resolvers).
 - This context is a child of the root context, allowing controllers to access service beans.
- DispatcherServlet Configuration:
 - The DispatcherServlet is created with the MVC context.
 - It's registered with the name "dispatcher" and mapped to the root URL (/), meaning it handles all incoming requests.
 - setLoadOnStartup(1) ensures the servlet is initialized when the application starts.

3. Root Application Context (AppConfig.java)

This configuration sets up the service layer and includes the JPA configuration.

```
package com.example.config;
```

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
```

```
@Configuration
@Import(JpaConfig.class) // Include JPA configuration
public class AppConfig {
    // Define service beans here if needed
}
```

- **Explanation:**

- @Import(JpaConfig.class) brings in the JPA configuration.
- This root context holds non-web beans (e.g., services, repositories).

4. JPA Configuration (JpaConfig.java)

Configure the EntityManagerFactory, DataSource, and Spring Data JPA repositories.

```
package com.example.config;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.orm.jpa.JpaTransactionManager;
```



```
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;

@Configuration
@EnableJpaRepositories(basePackages = "com.example.repository") // Enable Spring Data
JPA
@EnableTransactionManagement // Enable transaction management
public class JpaConfig {

    @Bean
    public DataSource dataSource() {
        // Use H2 for simplicity; replace with MySQL/PostgreSQL DataSource in production
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
            . setName("testdb")
            . build();
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean emf = new
        LocalContainerEntityManagerFactoryBean();
        emf.setDataSource(dataSource());
        emf.setPackagesToScan("com.example.entity"); // Scan for JPA entities
        emf.setJpaVendorAdapter(new urmeJpaVendorAdapter());
        emf.getJpaPropertyMap().put("hibernate.hbm2ddl.auto", "create-drop"); // Auto-create
        tables
        emf.getJpaPropertyMap().put("hibernate.dialect", "org.hibernate.dialect.H2Dialect");
        return emf;
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        JpaTransactionManager txManager = new JpaTransactionManager();
        txManager.setEntityManagerFactory(entityManagerFactory().getObjectType());
        return txManager;
    }
}
```

- **Explanation:**

- DataSource: Configures an embedded H2 database for simplicity. In production, use a DriverManagerDataSource or connection pool (e.g., HikariCP).



- LocalContainerEntityManagerFactoryBean: Sets up the JPA EntityManagerFactory with Hibernate as the provider.
- @EnableJpaRepositories: Enables Spring Data JPA repositories in the com.example.repository package.
- @EnableTransactionManagement: Enables annotation-driven transaction management.
- PlatformTransactionManager: Manages JPA transactions.

5. Spring MVC Configuration (WebConfig.java)

Configure MVC components like controllers and view resolvers.

```
package com.example.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.example.controller") // Scan for controllers
public class WebConfig implements WebMvcConfigurer {

    @Bean
    public InternalResourceViewResolver viewResolver() {
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/views/"); // Location of JSP files
        resolver.setSuffix(".jsp");
        return resolver;
    }
}
```

- **Explanation:**

- @EnableWebMvc: Enables Spring MVC.
- @ComponentScan: Scans for controllers in com.example.controller.
- InternalResourceViewResolver: Resolves JSP views (e.g., /WEB-INF/views/home.jsp).

List of Annotations Used and Their Roles

Below is a comprehensive list of annotations used in the spring-mvc-jpa project, along with their roles, based on the provided code (AppConfig.java, DispatcherServletInitializer.java, UserController.java, UserDAO.java, and entity classes).

Spring Annotations

1. **@Configuration** (in AppConfig.java):



- **Role:** Marks a class as a source of Spring bean definitions, replacing XML-based configuration.
 - **Usage:** Defines AppConfig as the configuration class for beans like DataSource, EntityManagerFactory, and ThymeleafViewResolver.
2. **@EnableWebMvc** (in AppConfig.java):
- **Role:** Enables Spring MVC configuration, activating annotations like @Controller, @RequestMapping, and MVC infrastructure (e.g., handler mappings).
 - **Usage:** Configures Spring MVC for handling web requests.
3. **@EnableTransactionManagement** (in AppConfig.java):
- **Role:** Enables Spring's annotation-driven transaction management, allowing @Transactional to create transactions.
 - **Usage:** Ensures @Transactional methods in UserDAO (e.g., createUser) run within a transaction.
4. **@ComponentScan(basePackages = "com.example")** (in AppConfig.java):
- **Role:** Scans the specified package (com.example) for Spring components (e.g., @Controller, @Repository, @Service).
 - **Usage:** Discovers UserController and UserDAO as Spring-managed beans.
5. **@Bean** (in AppConfig.java):
- **Role:** Defines a Spring-managed bean, created and managed by the Spring container.
 - **Usage:** Configures DataSource, LocalContainerEntityManagerFactoryBean, PlatformTransactionManager, ThymeleafViewResolver, etc.
6. **@Controller** (in UserController.java):
- **Role:** Marks a class as a Spring MVC controller, handling HTTP requests.
 - **Usage:** Identifies UserController as the component processing requests to /users.
7. **@RequestMapping("/users")** (in UserController.java):
- **Role:** Maps HTTP requests to controller methods or classes, specifying the base URL path.
 - **Usage:** Sets /users as the base path for all methods in UserController.
8. **@GetMapping, @PostMapping** (in UserController.java):
- **Role:** Maps HTTP GET or POST requests to specific controller methods.
 - **Usage:** Handles requests like GET /users, POST /users, GET /users/{id}, etc.
9. **@PathVariable("id")** (in UserController.java):
- **Role:** Binds a URL path variable (e.g., {id}) to a method parameter.
 - **Usage:** Extracts the id from URLs like /users/{id} for methods like showUser.
10. **@ModelAttribute** (in UserController.java):
- **Role:** Binds form data or request parameters to a model object (e.g., User, Address).
 - **Usage:** Populates User and Address objects from form submissions in createUser and updateUser.
11. **@Repository** (in UserDAO.java):
- **Role:** Marks a class as a data access object, enabling exception translation (e.g., converting JPA exceptions to Spring's DataAccessException).
 - **Usage:** Identifies UserDAO as a DAO, enabling Spring to manage it and apply transaction proxies.



12. @Transactional (in UserDao.java):

- **Role:** Declares that a method should run within a transaction, managed by Spring's transaction manager.
- **Usage:** Applied to createUser, updateUser, and deleteUser to ensure database operations are transactional.

13. @PersistenceContext (in UserDao.java):

- **Role:** Injects a JPA EntityManager into a Spring-managed bean, tied to the persistence context.
- **Usage:** Injects EntityManager into UserDao for database operations.

JPA/Hibernate Annotations**14. @Entity (in User.java, Address.java, Profile.java, Phone.java, Role.java):**

- **Role:** Marks a class as a JPA entity, mapping it to a database table.
- **Usage:** Defines User, Address, Profile, Phone, and Role as entities.

15. @Table (in entity classes):

- **Role:** Specifies the database table name for an entity.
- **Usage:** Maps User to a table (e.g., @Table(name = "users")).

16. @Id (in entity classes):

- **Role:** Marks a field as the primary key of an entity.
- **Usage:** Identifies the id field in User, Address, etc.

17. @GeneratedValue(strategy = GenerationType.IDENTITY) (in entity classes):

- **Role:** Specifies that the primary key is auto-generated by the database (e.g., auto-increment).
- **Usage:** Used on id fields to let MySQL generate unique IDs.

18. @Column (in entity classes):

- **Role:** Maps a field to a database column, optionally specifying properties like nullable or unique.
- **Usage:** Maps fields like name, email in User to columns.

19. @Transient (in entity classes):

- **Role:** Marks a field as non-persistent, excluding it from database mapping.
- **Usage:** Used for fields that should not be stored (e.g., computed properties).

20. @OneToOne (in User.java, Profile.java, Address.java):

- **Role:** Defines a one-to-one relationship between entities.
- **Usage:** Links User to Profile and Address.

21. @OneToMany (in User.java):

- **Role:** Defines a one-to-many relationship, where one entity is associated with multiple others.
- **Usage:** Links User to a list of Phone entities.

22. @ManyToOne (in Phone.java):

- **Role:** Defines a many-to-one relationship, where multiple entities relate to one parent.
- **Usage:** Links Phone to its owning User.

23. @ManyToMany (in User.java, Role.java):

- **Role:** Defines a many-to-many relationship, typically with a join table.
- **Usage:** Links User to multiple Role entities via a join table.

24. @Embeddable (in Address.java or similar):

- **Role:** Marks a class as embeddable, allowing its fields to be embedded in another entity.
 - **Usage:** Used if Address is embedded within User (optional, depending on your setup).
25. **@Embedded** (in User.java):
- **Role:** Specifies that an embeddable object's fields are included in the entity's table.
 - **Usage:** Embeds Address fields in the User table (if applicable).
26. **@NamedQuery, @NamedQueries** (in User.java):
- **Role:** Defines reusable JPQL queries for an entity.
 - **Usage:** Defines queries like User.findAllActive and User.findByEmail in User.
27. **@Inheritance** (in entity classes, if used):
- **Role:** Specifies the inheritance strategy for an entity hierarchy (e.g., SINGLE_TABLE, TABLE_PER_CLASS).
 - **Usage:** Used if entities like User have subclasses (not explicitly shown but possible).
28. **@DiscriminatorColumn** (in entity classes, if used):
- **Role:** Defines the column that distinguishes between subclasses in an inheritance hierarchy.
 - **Usage:** Used with @Inheritance to store a discriminator value.
29. **@EntityListeners** (in Auditable.java or entities):
- **Role:** Specifies a listener class for JPA lifecycle events (e.g., @PrePersist, @PostPersist).
 - **Usage:** Links AuditListener to entities for auditing creation/update timestamps.
30. **@PrePersist, @PostPersist** (in AuditListener.java):
- **Role:** Defines methods to be called before/after an entity is persisted.
 - **Usage:** Sets creation/update timestamps in Auditable entities.
31. **@Version** (in entity classes, if used):
- **Role:** Marks a field for optimistic locking, tracking entity versions.
 - **Usage:** Used for concurrency control (not explicitly shown but possible).

Database Relationships in Spring JPA - Complete Guide

What is a Database Relationship?

A **database relationship** defines how data in one table is connected to data in another table. Relationships help maintain data integrity, reduce redundancy, and establish logical connections between different entities in your database schema.

Types of Relationships

1. One-to-One (1:1) Relationship

Definition: Each record in Table A is associated with exactly one record in Table B, and vice versa.

Real-world Example: A person has one passport, and each passport belongs to one person.



Database Structure:

User Table:		UserProfile Table:		
		id	user_id	bio
1	john_doe	1	1	..."
2	jane_sm	2	2	..."

Spring JPA Implementation:

```
// Parent Entity (User)
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;

    // One-to-One relationship - User owns the relationship
    @OneToOne(mappedBy = "user", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    private UserProfile profile;

    // Getters and setters...
}
```

- **@Entity:** This annotation marks the class as an entity, meaning it will be mapped to a database table.
- **@Id:** This annotation specifies the primary key of the entity.
- **@GeneratedValue(strategy = GenerationType.IDENTITY):** This annotation indicates that the primary key will be automatically generated by the database.
- **private Long id:** The primary key field.
- **private String username:** A field representing the username.
- **@OneToOne:** This annotation defines a one-to-one relationship between User and UserProfile.
 - **mappedBy = "user":** This attribute indicates that the UserProfile entity is the owner of the relationship and that the User entity is mapped by the user field in the UserProfile entity.
 - **cascade = CascadeType.ALL:** This attribute specifies that all operations (persist, merge, refresh, remove) performed on the User entity should be cascaded to the UserProfile entity.
 - **fetch = FetchType.LAZY:** This attribute specifies that the UserProfile should be loaded lazily, meaning it will only be loaded when explicitly requested.

Child Entity (UserProfile)

```
@Entity
public class UserProfile {
```



```

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

// Foreign key reference to User
@OneToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "user_id", nullable = false)
private User user;

// Getters and setters...
}

```

- **@Entity:** This annotation marks the class as an entity.
- **@Id:** This annotation specifies the primary key of the entity.
- **@GeneratedValue(strategy = GenerationType.IDENTITY):** This annotation indicates that the primary key will be automatically generated by the database.
- **private Long id:** The primary key field.
- **private String bio:** A field representing the user's biography.
- **@OneToOne:** This annotation defines a one-to-one relationship between UserProfile and User.
 - **fetch = FetchType.LAZY:** This attribute specifies that the User should be loaded lazily.
- **@JoinColumn(name = "user_id", nullable = false):** This annotation specifies the foreign key column in the UserProfile table that references the primary key of the User table. The nullable = false attribute indicates that this column cannot be null.

2. One-to-Many (1:N) Relationship

Definition: Each record in Table A can be associated with multiple records in Table B, but each record in Table B is associated with only one record in Table A.

Real-world Example: A blog author can write many posts, but each post has only one author.

Database Structure:

User Table:		Post Table:		
+-----+	+-----+	+-----+	+-----+	+-----+
id	username	id	title	author_id
1	john_doe	1	"Post 1"	1
2	jane_sm	2	"Post 2"	1
+-----+	+-----+	+-----+	+-----+	+-----+
		3	"Post 3"	2
		+-----+	+-----+	+-----+

Spring JPA Implementation:

// Parent Entity (User) - One side

@Entity

public class User {



```
@Id  
@GeneratedValue(strategy = GenerationType.IDENTITY)  
private long id;  
private String username;  
  
// One-to-Many relationship  
@OneToMany(mappedBy = "author", cascade = CascadeType.ALL, fetch = FetchType.LAZY)  
private List<Post> posts = new ArrayList<>();  
  
// Helper methods  
public void addPost(Post post) {  
    posts.add(post);  
    post.setAuthor(this);  
}  
  
public void removePost(Post post) {  
    posts.remove(post);  
    post.setAuthor(null);  
}  
}
```

- `@OneToMany`: This annotation defines a one-to-many relationship between User and Post.
 - `mappedBy = "author"`: This attribute indicates that the Post entity is the owner of the relationship and that the User entity is mapped by the author field in the Post entity.
 - `cascade = CascadeType.ALL`: This attribute specifies that all operations (persist, merge, refresh, remove) performed on the User entity should be cascaded to the Post entities.
 - `fetch = FetchType.LAZY`: This attribute specifies that the Post entities should be loaded lazily, meaning they will only be loaded when explicitly requested.
- `private List<Post> posts = new ArrayList<>()`: A list to hold the Post entities associated with the User.

```
// Child Entity (Post) - Many side  
@Entity  
public class Post {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String title;  
  
    // Many-to-One relationship (reverse side of One-to-Many)  
    @ManyToOne(fetch = FetchType.LAZY)  
    @JoinColumn(name = "author_id", nullable = false)  
    private User author;
```



- ```
// Getters and setters...
}

 - @ManyToOne: This annotation defines a many-to-one relationship between Post and User.
 - fetch = FetchType.LAZY: This attribute specifies that the User should be loaded lazily.
 - @JoinColumn(name = "author_id", nullable = false): This annotation specifies the foreign key column in the Post table that references the primary key of the User table. The nullable = false attribute indicates that this column cannot be null.
 - private User author: A field representing the author of the post, which is a User entity.
```

### 3. Many-to-One (N:1) Relationship

**Definition:** Multiple records in Table A can be associated with one record in Table B.

**Note:** This is the reverse perspective of One-to-Many. In JPA, you typically use **@ManyToOne** on the "many" side.

**Real-world Example:** Many employees work in one department.

**Spring JPA Implementation:**

```
// Many side (Employee)
@Entity
public class Employee {
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
 private String name;

 // Many employees belong to one department
 @ManyToOne(fetch = FetchType.LAZY)
 @JoinColumn(name = "department_id")
 private Department department;
```

- ```
}
```
- **@ManyToOne:** This annotation defines a many-to-one relationship between Employee and Department.
 - **fetch = FetchType.LAZY:** This attribute specifies that the Department should be loaded lazily, meaning it will only be loaded when explicitly requested.
 - **@JoinColumn(name = "department_id"):** This annotation specifies the foreign key column in the Employee table that references the primary key of the Department table.
 - **private Department department:** A field representing the department to which the employee belongs.

```
// One side (Department)
@Entity
public class Department {
    @Id
```



```

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

    // One department has many employees
    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)
    private List<Employee> employees = new ArrayList<>();
}

```

- **@OneToMany:** This annotation defines a one-to-many relationship between Department and Employee.
 - **mappedBy = "department":** This attribute indicates that the Employee entity is the owner of the relationship and that the Department entity is mapped by the department field in the Employee entity.
 - **cascade = CascadeType.ALL:** This attribute specifies that all operations (persist, merge, refresh, remove) performed on the Department entity should be cascaded to the Employee entities.
- **private List<Employee> employees = new ArrayList<>():** A list to hold the Employee entities associated with the Department.

4. Many-to-Many (N:N) Relationship

Definition: Multiple records in Table A can be associated with multiple records in Table B, and vice versa.

Real-world Example: Students can enroll in multiple courses, and each course can have multiple students.

Database Structure:

User Table:	user_roles (Join Table):	Role Table:
+-----+ id username +-----+	+-----+-----+ user_id role_id +-----+-----+	+-----+-----+ id name +-----+-----+
1 john_doe	1 1	1 ADMIN
2 jane_sm	1 2	2 USER
+-----+	2 2	3 MODERATOR
	+-----+-----+	+-----+-----+

Spring JPA Implementation:

```

// First Entity (User)
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;

```



```

// Many-to-Many relationship - owning side
@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
@JoinTable(
    name = "user_roles",           // Join table name
    joinColumns = @JoinColumn(name = "user_id"),   // Foreign key to User
    inverseJoinColumns = @JoinColumn(name = "role_id") // Foreign key to Role
)
private Set<Role> roles = new HashSet<>();

// Helper methods to maintain relationship integrity
public void addRole(Role role) {
    roles.add(role);
    role.getUsers().add(this);
}

public void removeRole(Role role) {
    roles.remove(role);
    role.getUsers().remove(this);
}
}

```

- **@ManyToMany:** This annotation defines a many-to-many relationship between User and Role.
 - `cascade = {CascadeType.PERSIST, CascadeType.MERGE}:` This attribute specifies that persist and merge operations performed on the User entity should be cascaded to the Role entities.
- **@JoinTable:** This annotation specifies the join table used for the many-to-many relationship.
 - `name = "user_roles"`: The name of the join table.
 - `joinColumns = @JoinColumn(name = "user_id")`: Specifies the foreign key column in the join table that references the primary key of the User table.
 - `inverseJoinColumns = @JoinColumn(name = "role_id")`: Specifies the foreign key column in the join table that references the primary key of the Role table.
- `private Set<Role> roles = new HashSet<>()`: A set to hold the Role entities associated with the User.

```

// Second Entity (Role)
@Entity
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    // Many-to-Many relationship - non-owning side
    @ManyToMany(mappedBy = "roles")
    private Set<User> users = new HashSet<>();
}

```



```
// Getters and setters...
}



- @ManyToMany: This annotation defines a many-to-many relationship between Role and User.
  - mappedBy = "roles": This attribute indicates that the User entity is the owner of the relationship and that the Role entity is mapped by the roles field in the User entity.
- private Set<User> users = new HashSet<>();: A set to hold the User entities associated with the Role.

```

Advanced Relationship Concepts

1. Fetch Types

EAGER vs LAZY Loading:

```
// EAGER - Loads related entities immediately
@OneToMany(fetch = FetchType.EAGER)
private List<Post> posts;
```

```
// LAZY - Loads related entities on demand (default for collections)
@OneToMany(fetch = FetchType.LAZY)
private List<Post> posts;
```

When to use:

- **EAGER:** When you always need the related data
- **LAZY:** When you might not need the related data (recommended for performance)

2. Cascade Types

Controls how operations on parent entities affect related entities:

```
@OneToMany(cascade = CascadeType.ALL)
private List<Post> posts;
```

```
// Available cascade types:
// CascadeType.PERSIST - Save operations cascade
// CascadeType.MERGE - Update operations cascade
// CascadeType.REMOVE - Delete operations cascade
// CascadeType.REFRESH - Refresh operations cascade
// CascadeType.DETACH - Detach operations cascade
// CascadeType.ALL - All operations cascade
```

3. Orphan Removal

Automatically removes child entities when they're no longer referenced:

```
@OneToMany(mappedBy = "author", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Post> posts;
```

4. Bidirectional vs Unidirectional Relationships

Bidirectional: Both entities know about each other

```
// User entity
```



```
@OneToMany(mappedBy = "author")
private List<Post> posts;
```

```
// Post entity
@ManyToOne
@JoinColumn(name = "author_id")
private User author;
```

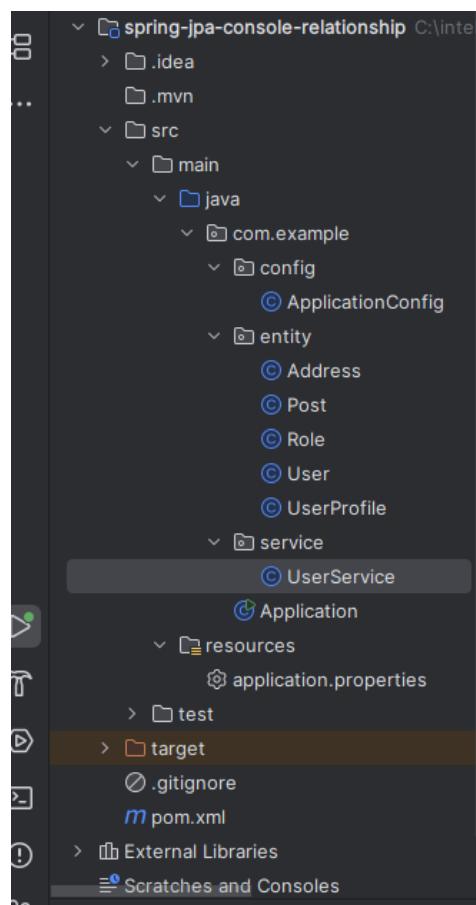
Unidirectional: Only one entity knows about the other

// Only Post knows about User

```
@ManyToOne
@JoinColumn(name = "author_id")
private User author;
```

Understanding these concepts will help you design efficient and maintainable database schemas with Spring JPA.

Console Application to Demonstrate some Above Listed Annotations and Relationships using Spring-JPA



The **Spring JPA CRUD Application** is a console-based Java application built using Spring Framework 6 and Java 21, designed to demonstrate basic Create, Read, Update, and Delete



(CRUD) operations with a relational database using Java Persistence API (JPA) and Hibernate. The application manages entities for users, their profiles, posts, and roles, showcasing JPA relationships including One-to-One, One-to-Many, Many-to-Many, and Embedded. It connects to a MySQL database (`jpa_example_db`) using HikariCP for connection pooling and provides a command-line interface for interacting with the data.

List of Files and Descriptions

Application.java: Main class providing a console-based menu interface for CRUD operations on users, posts, and roles using Spring.

```
package com.example;

import com.example.config.ApplicationConfig;
import com.example.entity.User;
import com.example.entity.Post;
import com.example.entity.Role;
import com.example.service.UserService;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import java.util.List;
import java.util.Scanner;

public class Application {
    public static void main(String[] args) {
        System.out.println("==> Spring 5 JPA CRUD Demo ==>\n");

        // Initialize Spring context
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext(ApplicationConfig.class);
        UserService userService = context.getBean(UserService.class);
        Scanner scanner = new Scanner(System.in);

        try {
            while (true) {
                System.out.println("\nMain Menu:");
                System.out.println("1. User Management");
                System.out.println("2. Post Management");
                System.out.println("3. Role Management");
                System.out.println("4. Exit");
                System.out.print("Enter your choice: ");

                int choice = scanner.nextInt();
                scanner.nextLine(); // Consume newline

                switch (choice) {
                    case 1:
                        userManagementMenu(userService, scanner);
                        break;
                    case 2:
                        postManagementMenu(userService, scanner);
                        break;
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void userManagementMenu(UserService userService, Scanner scanner) {
        System.out.println("User Management Menu:");
        System.out.println("1. Create User");
        System.out.println("2. Read User");
        System.out.println("3. Update User");
        System.out.println("4. Delete User");
        System.out.println("5. Back");
        System.out.print("Enter your choice: ");

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

        switch (choice) {
            case 1:
                createUser(userService, scanner);
                break;
            case 2:
                readUser(userService, scanner);
                break;
            case 3:
                updateUser(userService, scanner);
                break;
            case 4:
                deleteUser(userService, scanner);
                break;
            case 5:
                break;
        }
    }

    private void postManagementMenu(UserService userService, Scanner scanner) {
        System.out.println("Post Management Menu:");
        System.out.println("1. Create Post");
        System.out.println("2. Read Post");
        System.out.println("3. Update Post");
        System.out.println("4. Delete Post");
        System.out.println("5. Back");
        System.out.print("Enter your choice: ");

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

        switch (choice) {
            case 1:
                createPost(userService, scanner);
                break;
            case 2:
                readPost(userService, scanner);
                break;
            case 3:
                updatePost(userService, scanner);
                break;
            case 4:
                deletePost(userService, scanner);
                break;
            case 5:
                break;
        }
    }

    private void createUser(UserService userService, Scanner scanner) {
        System.out.print("Enter User Name: ");
        String name = scanner.nextLine();
        System.out.print("Enter User Email: ");
        String email = scanner.nextLine();
        System.out.print("Enter User Password: ");
        String password = scanner.nextLine();
        System.out.print("Enter User Role ID: ");
        int roleId = scanner.nextInt();
        scanner.nextLine();

        User user = new User(name, email, password, roleId);
        userService.createUser(user);
        System.out.println("User created successfully!");
    }

    private void readUser(UserService userService, Scanner scanner) {
        System.out.print("Enter User ID: ");
        int userId = scanner.nextInt();
        scanner.nextLine();

        User user = userService.readUser(userId);
        if (user != null) {
            System.out.println("User found: " + user);
        } else {
            System.out.println("User not found.");
        }
    }

    private void updateUser(UserService userService, Scanner scanner) {
        System.out.print("Enter User ID: ");
        int userId = scanner.nextInt();
        scanner.nextLine();

        User user = userService.readUser(userId);
        if (user != null) {
            System.out.print("Enter New User Name: ");
            String name = scanner.nextLine();
            System.out.print("Enter New User Email: ");
            String email = scanner.nextLine();
            System.out.print("Enter New User Password: ");
            String password = scanner.nextLine();
            System.out.print("Enter New User Role ID: ");
            int roleId = scanner.nextInt();
            scanner.nextLine();

            user.setName(name);
            user.setEmail(email);
            user.setPassword(password);
            user.setRoleId(roleId);
            userService.updateUser(user);
            System.out.println("User updated successfully!");
        } else {
            System.out.println("User not found.");
        }
    }

    private void deleteUser(UserService userService, Scanner scanner) {
        System.out.print("Enter User ID: ");
        int userId = scanner.nextInt();
        scanner.nextLine();

        User user = userService.readUser(userId);
        if (user != null) {
            userService.deleteUser(user);
            System.out.println("User deleted successfully!");
        } else {
            System.out.println("User not found.");
        }
    }

    private void createPost(UserService userService, Scanner scanner) {
        System.out.print("Enter Post Title: ");
        String title = scanner.nextLine();
        System.out.print("Enter Post Content: ");
        String content = scanner.nextLine();
        System.out.print("Enter Post User ID: ");
        int userId = scanner.nextInt();
        scanner.nextLine();

        Post post = new Post(title, content, userId);
        userService.createPost(post);
        System.out.println("Post created successfully!");
    }

    private void readPost(UserService userService, Scanner scanner) {
        System.out.print("Enter Post ID: ");
        int postId = scanner.nextInt();
        scanner.nextLine();

        Post post = userService.readPost(postId);
        if (post != null) {
            System.out.println("Post found: " + post);
        } else {
            System.out.println("Post not found.");
        }
    }

    private void updatePost(UserService userService, Scanner scanner) {
        System.out.print("Enter Post ID: ");
        int postId = scanner.nextInt();
        scanner.nextLine();

        Post post = userService.readPost(postId);
        if (post != null) {
            System.out.print("Enter New Post Title: ");
            String title = scanner.nextLine();
            System.out.print("Enter New Post Content: ");
            String content = scanner.nextLine();
            System.out.print("Enter New Post User ID: ");
            int userId = scanner.nextInt();
            scanner.nextLine();

            post.setTitle(title);
            post.setContent(content);
            post.setUserId(userId);
            userService.updatePost(post);
            System.out.println("Post updated successfully!");
        } else {
            System.out.println("Post not found.");
        }
    }

    private void deletePost(UserService userService, Scanner scanner) {
        System.out.print("Enter Post ID: ");
        int postId = scanner.nextInt();
        scanner.nextLine();

        Post post = userService.readPost(postId);
        if (post != null) {
            userService.deletePost(post);
            System.out.println("Post deleted successfully!");
        } else {
            System.out.println("Post not found.");
        }
    }
}
```



```
case 3:  
    roleManagementMenu(userService, scanner);  
    break;  
case 4:  
    System.out.println("Exiting...");  
    scanner.close();  
    context.close();  
    return;  
default:  
    System.out.println("Invalid choice. Please try again.");  
}  
}  
}  
} catch (Exception e) {  
    System.err.println("Error occurred: " + e.getMessage());  
    e.printStackTrace();  
} finally {  
    scanner.close();  
    context.close();  
}  
}  
  
private static void userManagementMenu(UserService userService, Scanner scanner) {  
    while (true) {  
        System.out.println("\nUser Management:");  
        System.out.println("1. Create User");  
        System.out.println("2. Read User");  
        System.out.println("3. Update User");  
        System.out.println("4. Delete User");  
        System.out.println("5. List All Users");  
        System.out.println("6. Back to Main Menu");  
        System.out.print("Enter your choice: ");  
  
        int choice = scanner.nextInt();  
        scanner.nextLine(); // Consume newline  
  
        switch (choice) {  
            case 1:  
                System.out.print("Enter username: ");  
                String username = scanner.nextLine();  
                System.out.print("Enter email: ");  
                String email = scanner.nextLine();  
                System.out.print("Enter first name: ");  
                String firstName = scanner.nextLine();  
                System.out.print("Enter last name: ");  
                String lastName = scanner.nextLine();  
                System.out.print("Enter phone number: ");  
                String phoneNumber = scanner.nextLine();  
                System.out.print("Enter bio: ");  
                String bio = scanner.nextLine();  
                System.out.print("Enter street address: ");  
                String street = scanner.nextLine();  
        }  
    }  
}
```



```
System.out.print("Enter city: ");
String city = scanner.nextLine();
System.out.print("Enter state: ");
String state = scanner.nextLine();
System.out.print("Enter zip code: ");
String zipCode = scanner.nextLine();
User user = userService.createUserWithProfile(
    username, email, firstName, lastName, phoneNumber, bio,
    street, city, state, zipCode
);
System.out.println("Created: " + user);
break;
case 2:
    System.out.print("Enter user ID: ");
    Long userId = scanner.nextLong();
    scanner.nextLine(); // Consume newline
    User foundUser = userService.findUserById(userId);
    if (foundUser != null) {
        System.out.println("Found: " + foundUser);
    } else {
        System.out.println("User not found");
    }
    break;
case 3:
    System.out.print("Enter user ID: ");
    Long updateUserId = scanner.nextLong();
    scanner.nextLine(); // Consume newline
    System.out.print("Enter new email: ");
    String newEmail = scanner.nextLine();
    userService.updateUserEmail(updateUserId, newEmail);
    System.out.println("User updated");
    break;
case 4:
    System.out.print("Enter user ID: ");
    Long deleteUserId = scanner.nextLong();
    scanner.nextLine(); // Consume newline
    userService.deleteUser(deleteUserId);
    System.out.println("User deleted");
    break;
case 5:
    List<User> allUsers = userService.findAllUsers();
    System.out.println("All Users:");
    for (User u : allUsers) {
        System.out.println(" " + u);
    }
    break;
case 6:
    return;
default:
    System.out.println("Invalid choice. Please try again.");
}
```



```
    }

}

private static void postManagementMenu(UserService userService, Scanner scanner) {
    while (true) {
        System.out.println("\nPost Management:");
        System.out.println("1. Create Post");
        System.out.println("2. Read Post");
        System.out.println("3. Update Post");
        System.out.println("4. Delete Post");
        System.out.println("5. List All Posts");
        System.out.println("6. Back to Main Menu");
        System.out.print("Enter your choice: ");

        int choice = scanner.nextInt();
        scanner.nextLine(); // Consume newline

        switch (choice) {
            case 1:
                System.out.print("Enter user ID: ");
                long userId = scanner.nextLong();
                scanner.nextLine(); // Consume newline
                System.out.print("Enter post title: ");
                String title = scanner.nextLine();
                System.out.print("Enter post content: ");
                String content = scanner.nextLine();
                Post post = userService.createPost(userId, title, content);
                System.out.println("Created: " + post);
                break;
            case 2:
                System.out.print("Enter post ID: ");
                Long postId = scanner.nextLong();
                scanner.nextLine(); // Consume newline
                Post foundPost = userService.findPostById(postId);
                if (foundPost != null) {
                    System.out.println("Found: " + foundPost);
                } else {
                    System.out.println("Post not found");
                }
                break;
            case 3:
                System.out.print("Enter post ID: ");
                Long updatePostId = scanner.nextLong();
                scanner.nextLine(); // Consume newline
                System.out.print("Enter new title: ");
                String newTitle = scanner.nextLine();
                System.out.print("Enter new content: ");
                String newContent = scanner.nextLine();
                userService.updatePost(updatePostId, newTitle, newContent);
                System.out.println("Post updated");
                break;
        }
    }
}
```



```
case 4:  
    System.out.print("Enter post ID: ");  
    Long deletePostId = scanner.nextLong();  
    scanner.nextLine(); // Consume newline  
    userService.deletePost(deletePostId);  
    System.out.println("Post deleted");  
    break;  
case 5:  
    List<Post> allPosts = userService.findAllPosts();  
    System.out.println("All Posts:");  
  
    for (Post p : allPosts) {  
        System.out.println(" " + p);  
    }  
    break;  
case 6:  
    return;  
default:  
    System.out.println("Invalid choice. Please try again.");  
}  
}  
}  
  
private static void roleManagementMenu(UserService userService, Scanner scanner) {  
    while (true) {  
        System.out.println("\nRole Management:");  
        System.out.println("1. Create Role");  
        System.out.println("2. Read Role");  
        System.out.println("3. Update Role");  
        System.out.println("4. Delete Role");  
        System.out.println("5. List All Roles");  
        System.out.println("6. Back to Main Menu");  
        System.out.print("Enter your choice: ");  
  
        int choice = scanner.nextInt();  
        scanner.nextLine(); // Consume newline  
  
        switch (choice) {  
            case 1:  
                System.out.print("Enter role name: ");  
                String name = scanner.nextLine();  
                System.out.print("Enter role description: ");  
                String description = scanner.nextLine();  
                Role role = userService.createRole(name, description);  
                System.out.println("Created: " + role);  
                break;  
            case 2:  
                System.out.print("Enter role ID: ");  
                Long roleId = scanner.nextLong();  
                scanner.nextLine(); // Consume newline
```



```
Role foundRole = userService.findRoleById(roleId);
if (foundRole != null) {
    System.out.println("Found: " + foundRole);
} else {
    System.out.println("Role not found");
}
break;
case 3:
    System.out.print("Enter role ID: ");
    Long updateRoleId = scanner.nextLong();
    scanner.nextLine(); // Consume newline
    System.out.print("Enter new role name: ");
    String newName = scanner.nextLine();
    System.out.print("Enter new role description: ");
    String newDescription = scanner.nextLine();
    userService.updateRole(updateRoleId, newName, newDescription);
    System.out.println("Role updated");
    break;
case 4:
    System.out.print("Enter role ID: ");
    Long deleteRoleId = scanner.nextLong();
    scanner.nextLine(); // Consume newline
    userService.deleteRole(deleteRoleId);
    System.out.println("Role deleted");
    break;
case 5:
    List<Role> allRoles = userService.findAllRoles();
    System.out.println("All Roles:");
    for (Role r : allRoles) {
        System.out.println(" " + r);
    }
    break;
case 6:
    return;
default:
    System.out.println("Invalid choice. Please try again.");
}
}
}
}
```

ApplicationConfig.java: Spring configuration file setting up the HikariCP data source, JPA with Hibernate, and transaction management.

```
package com.example.config;

import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
```



```
import org.springframework.context.annotation.Configuration;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;
import java.util.Properties;

@Configuration
@ComponentScan(basePackages = "com.example")
@EnableTransactionManagement
public class ApplicationConfig {

    @Bean
    public DataSource dataSource() {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:mysql://localhost:3306/jpa_example_db");
        config.setUsername("root");
        config.setPassword("Archer@12345");
        config.setDriverClassName("com.mysql.cj.jdbc.Driver");
        config.setMaximumPoolSize(10);
        config.setMinimumIdle(5);
        config.setConnectionTimeout(30000);
        config.setIdleTimeout(600000);
        config.setMaxLifetime(1800000);

        return new HikariDataSource(config);
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean emf = new
        LocalContainerEntityManagerFactoryBean();
        emf.setDataSource(dataSource());
        emf.setPackagesToScan("com.example.entity");
        emf.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
        emf.setJpaProperties(jpaProperties());

        return emf;
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        JpaTransactionManager txManager = new JpaTransactionManager();
        txManager.setEntityManagerFactory(entityManagerFactory().getObject());
        return txManager;
    }

    private Properties jpaProperties() {
```



```
Properties properties = new Properties();
properties.put("hibernate.dialect", "org.hibernate.dialect.MySQL8Dialect");
properties.put("hibernate.hbm2ddl.auto", "update");
properties.put("hibernate.show_sql", "true");
properties.put("hibernate.format_sql", "true");

return properties;
}
}
```

UserService.java: Service class implementing CRUD operations for User, Post, and Role entities using JPA EntityManager.

```
package com.example.service;

import com.example.entity.User;
import com.example.entity.UserProfile;
import com.example.entity.Post;
import com.example.entity.Role;
import com.example.entity.Address;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.persistence.TypedQuery;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service
@Transactional
public class UserService {

    @PersistenceContext
    private EntityManager entityManager;

    // User methods
    public User createUser(String username, String email) {
        User user = new User(username, email);
        entityManager.persist(user);
        return user;
    }

    public User createUserWithProfile(String username, String email,
                                    String firstName, String lastName,
                                    String phoneNumber, String bio,
                                    String street, String city, String state, String zipCode) {
        User user = new User(username, email);
        UserProfile profile = new UserProfile(firstName, lastName, phoneNumber, bio);
        Address address = new Address(street, city, state, zipCode);
        profile.setUser(user);
        user.setProfile(profile);
    }
}
```



```
user.setAddress(address);
entityManager.persist(user);
entityManager.persist(profile);
return user;
}

public User addAddressToUser(Long userId, String street, String city, String state, String zipCode) {
    User user = entityManager.find(User.class, userId);
    if (user != null) {
        Address address = new Address(street, city, state, zipCode);
        user.setAddress(address);
        entityManager.merge(user);
    }
    return user;
}

@Transactional(readOnly = true)
public User findUserById(Long id) {
    return entityManager.find(User.class, id);
}

@Transactional(readOnly = true)
public User findUserByUsername(String username) {
    TypedQuery<User> query = entityManager.createQuery(
        "SELECT u FROM User u WHERE u.username = :username", User.class);
    query.setParameter("username", username);
    List<User> users = query.getResultList();
    return users.isEmpty() ? null : users.get(0);
}

@Transactional(readOnly = true)
public List<User> findAllUsers() {
    TypedQuery<User> query = entityManager.createQuery(
        "SELECT u FROM User u", User.class);
    return query.getResultList();
}

public void updateUserEmail(Long userId, String newEmail) {
    User user = entityManager.find(User.class, userId);
    if (user != null) {
        user.setEmail(newEmail);
        entityManager.merge(user);
    }
}

public void deleteUser(Long id) {
    User user = entityManager.find(User.class, id);
    if (user != null) {
        entityManager.remove(user);
    }
}
```



```
// Post methods
public Post createPost(Long userId, String title, String content) {
    // Fetch the User within the session to ensure it's managed
    User user = entityManager.find(User.class, userId);
    if (user == null) {
        throw new IllegalArgumentException("User with ID " + userId + " not found");
    }
    Post post = new Post(title, content, user);
    entityManager.persist(post);
    return post;
}

@Transactional(readOnly = true)
public Post findPostById(Long id) {
    // Use a query with JOIN FETCH to eagerly load the User (author) to avoid
LazyInitializationException
    TypedQuery<Post> query = entityManager.createQuery(
        "SELECT p FROM Post p LEFT JOIN FETCH p.author WHERE p.id = :id", Post.class);
    query.setParameter("id", id);
    List<Post> posts = query.getResultList();
    return posts.isEmpty() ? null : posts.get(0);
}

@Transactional(readOnly = true)
public List<Post> findAllPosts() {
    // Eagerly fetch the User (author) for all posts to ensure safe access outside the session
    TypedQuery<Post> query = entityManager.createQuery(
        "SELECT p FROM Post p LEFT JOIN FETCH p.author", Post.class);
    return query.getResultList();
}

public void updatePost(Long postId, String newTitle, String newContent) {
    Post post = entityManager.find(Post.class, postId);
    if (post == null) {
        throw new IllegalArgumentException("Post with ID " + postId + " not found");
    }
    post.setTitle(newTitle);
    post.setContent(newContent);
    // No need to call merge() since the entity is already managed in the transactional context
}

public void deletePost(Long id) {
    Post post = entityManager.find(Post.class, id);
    if (post != null) {
        entityManager.remove(post);
    } else {
        throw new IllegalArgumentException("Post with ID " + id + " not found");
    }
}
```



```
@Transactional(readOnly = true)
public List<Post> findPostsByUser(Long userId) {
    // Eagerly fetch the User (author) to avoid LazyInitializationException
    TypedQuery<Post> query = entityManager.createQuery(
        "SELECT p FROM Post p LEFT JOIN FETCH p.author WHERE p.author.id = :userId ORDER BY
        p.createdAt DESC",
        Post.class);
    query.setParameter("userId", userId);
    return query.getResultList();
}

// Role methods
public Role createRole(String name, String description) {
    Role role = new Role(name, description);
    entityManager.persist(role);
    return role;
}

@Transactional(readOnly = true)
public Role findRoleById(Long id) {
    return entityManager.find(Role.class, id);
}

@Transactional(readOnly = true)
public List<Role> findAllRoles() {
    TypedQuery<Role> query = entityManager.createQuery(
        "SELECT r FROM Role r", Role.class);
    return query.getResultList();
}

public void updateRole(Long roleId, String newName, String newDescription) {
    Role role = entityManager.find(Role.class, roleId);
    if (role != null) {
        role.setName(newName);
        role.setDescription(newDescription);
        entityManager.merge(role);
    }
}

public void deleteRole(Long id) {
    Role role = entityManager.find(Role.class, id);
    if (role != null) {
        entityManager.remove(role);
    }
}

public void assignRoleToUser(Long userId, Long roleId) {
    User user = entityManager.find(User.class, userId);
    Role role = entityManager.find(Role.class, roleId);
    if (user != null && role != null) {
        user.addRole(role);
    }
}
```



```
        entityManager.merge(user);
    }
}
}
```

User.java: Entity class representing a user with fields for username, email, address (embedded), profile, posts, and roles.

```
package com.example.entity;

import jakarta.persistence.*;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

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

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

    @Column(name = "username", unique = true, nullable = false)
    private String username;

    @Column(name = "email", nullable = false)
    private String email;

    @Column(name = "created_at")
    private LocalDateTime createdAt;

    @Transient
    private String temporaryToken;

    @Embedded
    private Address address;

    @OneToOne(mappedBy = "user", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    private UserProfile profile;

    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    private List<Post> posts = new ArrayList<>();

    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinTable(
        name = "user_roles",
        joinColumns = @JoinColumn(name = "user_id"),
        inverseJoinColumns = @JoinColumn(name = "role_id")
    )
}
```



```
)  
private Set<Role> roles = new HashSet<>();  
  
public User() {}  
  
public User(String username, String email) {  
    this.username = username;  
    this.email = email;  
    this.createdAt = LocalDateTime.now();  
}  
  
// 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 getEmail() { return email; }  
public void setEmail(String email) { this.email = email; }  
  
public LocalDateTime getCreatedAt() { return createdAt; }  
public void setCreatedAt(LocalDateTime createdAt) { this.createdAt = createdAt; }  
  
public String getTemporaryToken() { return temporaryToken; }  
public void setTemporaryToken(String temporaryToken) { this.temporaryToken = temporaryToken; }  
  
public Address getAddress() { return address; }  
public void setAddress(Address address) { this.address = address; }  
  
public UserProfile getProfile() { return profile; }  
public void setProfile(UserProfile profile) { this.profile = profile; }  
  
public List<Post> getPosts() { return posts; }  
public void setPosts(List<Post> posts) { this.posts = posts; }  
  
public Set<Role> getRoles() { return roles; }  
public void setRoles(Set<Role> roles) { this.roles = roles; }  
  
public void addRole(Role role) {  
    roles.add(role);  
    role.getUsers().add(this);  
}  
  
public void removeRole(Role role) {  
    roles.remove(role);  
    role.getUsers().remove(this);  
}  
  
@Override  
public String toString() {
```



```
        return String.format("User{id=%d, username='%s', email='%s'}", id, username, email);
    }
}
```

UserProfile.java: Entity class defining a user's profile with fields like first name, last name, phone number, and bio.

```
package com.example.entity;
```

```
import jakarta.persistence.*;
```

```
@Entity
```

```
@Table(name = "user_profiles")
```

```
public class UserProfile {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    @Column(name = "first_name")
```

```
    private String firstName;
```

```
    @Column(name = "last_name")
```

```
    private String lastName;
```

```
    @Column(name = "phone_number")
```

```
    private String phoneNumber;
```

```
    @Column(name = "bio", columnDefinition = "TEXT")
```

```
    private String bio;
```

```
    @OneToOne(fetch = FetchType.LAZY)
```

```
    @JoinColumn(name = "user_id", nullable = false)
```

```
    private User user;
```

```
    public UserProfile() {}
```

```
    public UserProfile(String firstName, String lastName, String phoneNumber, String bio) {
```

```
        this.firstName = firstName;
```

```
        this.lastName = lastName;
```

```
        this.phoneNumber = phoneNumber;
```

```
        this.bio = bio;
```

```
}
```

```
    // Getters and Setters
```

```
    public Long getId() { return id; }
```

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

```
    public String getFirstName() { return firstName; }
```

```
    public void setFirstName(String firstName) { this.firstName = firstName; }
```



```
public String getLastName() { return lastName; }
public void setLastName(String lastName) { this.lastName = lastName; }

public String getPhoneNumber() { return phoneNumber; }
public void setPhoneNumber(String phoneNumber) { this.phoneNumber = phoneNumber; }

public String getBio() { return bio; }
public void setBio(String bio) { this.bio = bio; }

public User getUser() { return user; }
public void setUser(User user) { this.user = user; }

@Override
public String toString() {
    return String.format("UserProfile{id=%d, firstName='%s', lastName='%s'}",
        id, firstName, lastName);
}
}
```

Post.java: Entity class for posts, including title, content, creation timestamp, view count, and author (user).

```
package com.example.entity;

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

@Entity
@Table(name = "posts")
public class Post {

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

    @Column(name = "title", nullable = false)
    private String title;

    @Column(name = "content", columnDefinition = "TEXT")
    private String content;

    @Column(name = "created_at")
    private LocalDateTime createdAt;

    @Column(name = "view_count")
    private Integer viewCount = 0;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "author_id", nullable = false)
    private User author;
```



```
@Transient
private boolean isNew;

public Post() {}

public Post(String title, String content, User author) {
    this.title = title;
    this.content = content;
    this.author = author;
    this.createdAt = LocalDateTime.now();
    this.isNew = true;
}

// Getters and Setters
public Long getId() { return id; }
public void setId(Long id) { this.id = id; }

public String getTitle() { return title; }
public void setTitle(String title) { this.title = title; }

public String getContent() { return content; }
public void setContent(String content) { this.content = content; }

public LocalDateTime getCreatedAt() { return createdAt; }
public void setCreatedAt(LocalDateTime createdAt) { this.createdAt = createdAt; }

public Integer getViewCount() { return viewCount; }
public void setViewCount(Integer viewCount) { this.viewCount = viewCount; }

public User getAuthor() { return author; }
public void setAuthor(User author) { this.author = author; }

public boolean isNew() { return isNew; }
public void setNew(boolean isNew) { this.isNew = isNew; }

@Override
public String toString() {
    return String.format("Post{id=%d, title='%s', author='%s'}",
        id, title, author != null ? author.getUsername() : "Unknown");
}

public void setTemporaryToken(String s) {
}

public String getTemporaryToken() {
    return "";
}
}
```

Role.java Entity class for roles with name and description, linked to users via a many-to-many relationship.



```
package com.example.entity;

import jakarta.persistence.*;
import java.util.HashSet;
import java.util.Set;

@Entity
@Table(name = "roles")
public class Role {

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

    @Column(name = "name", unique = true, nullable = false)
    private String name;

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

    @ManyToMany(mappedBy = "roles")
    private Set<User> users = new HashSet<>();

    public Role() {}

    public Role(String name, String description) {
        this.name = name;
        this.description = description;
    }

    // Getters and Setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

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

    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }

    public Set<User> getUsers() { return users; }
    public void setUsers(Set<User> users) { this.users = users; }

    @Override
    public String toString() {
        return String.format("Role{id=%d, name='%s'}", id, name);
    }
}
```

Address.java: Embeddable class for user addresses, mapping to columns (street, city, state, zip code) in the users table.



```
package com.example.entity;

import jakarta.persistence.Column;
import jakarta.persistence.Embeddable;

@Embeddable
public class Address {

    @Column(name = "street_address")
    private String street;

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

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

    @Column(name = "zip_code")
    private String zipCode;

    public Address() {}

    public Address(String street, String city, String state, String zipCode) {
        this.street = street;
        this.city = city;
        this.state = state;
        this.zipCode = zipCode;
    }

    // Getters and Setters
    public String getStreet() { return street; }
    public void setStreet(String street) { this.street = street; }

    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }

    public String getState() { return state; }
    public void setState(String state) { this.state = state; }

    public String getZipCode() { return zipCode; }
    public void setZipCode(String zipCode) { this.zipCode = zipCode; }

    @Override
    public String toString() {
        return String.format("%s, %s, %s %s", street, city, state, zipCode);
    }
}
```

List of Tables and Relationships

1. users

- **Description:** Stores user information, including embedded address fields.



- **Columns** (based on User.java and embedded Address.java):
 - id (BIGINT, PRIMARY KEY, AUTO_INCREMENT)
 - username (VARCHAR, UNIQUE, NOT NULL)
 - email (VARCHAR, NOT NULL)
 - created_at (DATETIME)
 - street_address (VARCHAR, from Address)
 - city (VARCHAR, from Address)
 - state (VARCHAR, from Address)
 - zip_code (VARCHAR, from Address)
- **Relationships:**
 - Contains embedded Address fields.
 - One-to-One with user_profiles (via user_id in user_profiles).
 - One-to-Many with posts (via author_id in posts).
 - Many-to-Many with roles (via user_roles join table).

2. user_profiles

- **Description:** Stores user profile details, linked to a user.
- **Columns** (based on UserProfile.java):
 - id (BIGINT, PRIMARY KEY, AUTO_INCREMENT)
 - user_id (BIGINT, FOREIGN KEY to users.id, NOT NULL)
 - first_name (VARCHAR)
 - last_name (VARCHAR)
 - phone_number (VARCHAR)
 - bio (TEXT)
- **Relationships:**
 - One-to-One with users (owns the relationship via user_id).

3. posts

- **Description:** Stores post details, each linked to a user as the author.
- **Columns** (based on Post.java):
 - id (BIGINT, PRIMARY KEY, AUTO_INCREMENT)
 - title (VARCHAR, NOT NULL)
 - content (TEXT)
 - created_at (DATETIME)
 - view_count (INTEGER, DEFAULT 0)
 - author_id (BIGINT, FOREIGN KEY to users.id, NOT NULL)
- **Relationships:**
 - Many-to-One with users (via author_id).

4. roles

- **Description:** Stores role information.
- **Columns** (based on Role.java):
 - id (BIGINT, PRIMARY KEY, AUTO_INCREMENT)
 - name (VARCHAR, UNIQUE, NOT NULL)
 - description (VARCHAR)
- **Relationships:**
 - Many-to-Many with users (via user_roles join table).

5. user_roles

- **Description:** Join table for the many-to-many relationship between users and roles.



- **Columns** (based on @JoinTable in User.java):
 - user_id (BIGINT, FOREIGN KEY to users.id)
 - role_id (BIGINT, FOREIGN KEY to roles.id)
- **Relationships:**
 - Links users and roles for the many-to-many relationship.

SQL Queries to Demonstrate Relationships

Below are SQL queries that use JOINS to fetch related data, proving the relationships between tables. Each query includes a brief explanation of how it demonstrates the relationship.

-- Query 1: User ↔ UserProfile (One-to-One)

-- Purpose: Join users and user_profiles to show each user has exactly one profile

```
SELECT u.id, u.username, u.email, p.first_name, p.last_name, p.phone_number, p.bio  
FROM users u  
INNER JOIN user_profiles p ON u.id = p.user_id;
```

-- Query 2: User ↔ Address (Embedded)

-- Purpose: Select address fields from users table to show embedded address data

```
SELECT id, username, street_address, city, state, zip_code  
FROM users  
WHERE street_address IS NOT NULL;
```

-- Query 3: User ↔ Post (One-to-Many)

-- Purpose: Join users and posts to show all posts authored by each user

```
SELECT u.id, u.username, p.id AS post_id, p.title, p.content, p.created_at  
FROM users u  
LEFT JOIN posts p ON u.id = p.author_id  
ORDER BY u.id, p.created_at DESC;
```

-- Query 4: User ↔ Role (Many-to-Many)

-- Purpose: Join users, user_roles, and roles to show which roles are assigned to each user

```
SELECT u.id, u.username, r.id AS role_id, r.name AS role_name, r.description  
FROM users u  
INNER JOIN user_roles ur ON u.id = ur.user_id  
INNER JOIN roles r ON ur.role_id = r.id;
```

-- Query 5: Combined Query (All Relationships)

-- Purpose: Join all tables to show users with their profiles, addresses, posts, and roles

```
SELECT u.id, u.username, u.email, u.street_address, u.city, u.state, u.zip_code,  
      p.first_name, p.last_name, p.phone_number,  
      po.id AS post_id, po.title AS post_title,  
      r.id AS role_id, r.name AS role_name  
  FROM users u  
INNER JOIN user_profiles p ON u.id = p.user_id  
LEFT JOIN posts po ON u.id = po.author_id  
LEFT JOIN user_roles ur ON u.id = ur.user_id
```



```
LEFT JOIN roles r ON ur.role_id = r.id  
ORDER BY u.id, po.created_at DESC, r.id;
```

JPA CRUD Application, JPA (Java Persistence API) and Hibernate play critical roles in managing data persistence and interactions with the MySQL database (`jpa_example_db`). Below, I outline their roles in the context of the provided application, focusing on how they are utilized in the code to handle entity management, relationships, and database operations.

Role of JPA

JPA is a standard Java specification (part of Jakarta EE) that provides an API for object-relational mapping (ORM), allowing developers to map Java objects to database tables and perform CRUD operations using a high-level, object-oriented approach.

1. Entity Mapping:

- **Purpose:** JPA defines how Java classes (`User`, `UserProfile`, `Post`, `Role`, `Address`) are mapped to database tables (`users`, `user_profiles`, `posts`, `roles`, `user_roles`).
- **Implementation:** Annotations like `@Entity`, `@Table`, `@Id`, `@Column`, `@Embedded`, `@OneToOne`, `@OneToMany`, and `@ManyToMany` in the entity classes (`User.java`, `UserProfile.java`, `Post.java`, `Role.java`, `Address.java`) specify table names, column mappings, and relationships.
 - Example: In `User.java`, `@Entity` and `@Table(name = "users")` map the `User` class to the `users` table, while `@Embedded` maps `Address` fields as columns in the `users` table.
 - Example: In `UserProfile.java`, `@OneToOne` with `@JoinColumn(name = "user_id")` establishes a one-to-one relationship with `User`.

2. Relationship Management:

- **Purpose:** JPA manages relationships between entities (One-to-One, One-to-Many, Many-to-Many, Embedded).
- **Implementation:**
 - **One-to-One:** `User` ↔ `UserProfile` via `user_id` foreign key in `user_profiles`.
 - **Embedded:** `Address` fields (`street_address`, `city`, `state`, `zip_code`) are embedded in the `users` table.
 - **One-to-Many:** `User` ↔ `Post` via `author_id` in `posts`.
 - **Many-to-Many:** `User` ↔ `Role` via the `user_roles` join table, defined by `@JoinTable` in `User.java`.
- **Example:** The `User.addRole(Role)` method in `User.java` maintains bidirectional consistency for the many-to-many relationship.

3. CRUD Operations:

- **Purpose:** JPA provides a standard API for persisting, retrieving, updating, and deleting entities.
- **Implementation:** The `UserService.java` class uses `EntityManager` (a JPA interface) to perform operations like `persist`, `find`, `merge`, and `remove`.
 - Example: `createUser` persists a new `User` entity, and `findUserById` retrieves a user by ID.
 - Example: `createPost` persists a `Post` entity with a reference to a `User` via `author_id`.



4. Query Language (JPQL):

- **Purpose:** JPA allows querying the database using Java Persistence Query Language (JPQL), which operates on entities rather than raw SQL.
- **Implementation:** Methods like `findUserByUsername` and `findPostsByUser` in `UserService.java` use JPQL queries to retrieve data.
 - Example: `SELECT u FROM User u WHERE u.username = :username` retrieves a user by username.
 - Example: `SELECT p FROM Post p WHERE p.author.id = :userId ORDER BY p.createdAt DESC` fetches posts for a specific user.

5. Transaction Management:

- **Purpose:** JPA integrates with Spring's transaction management to ensure data consistency.
- **Implementation:** The `@Transactional` annotation in `UserService.java` ensures operations like `persist` and `merge` occur within a transaction, as configured in `ApplicationConfig.java` with `JpaTransactionManager`.

Role of Hibernate

Hibernate is an implementation of the JPA specification and serves as the underlying ORM framework in the application, providing additional features beyond the JPA standard.

1. JPA Provider:

- **Purpose:** Hibernate acts as the JPA provider, implementing the `EntityManager` interface used in `UserService.java`.
- **Implementation:** Configured in `ApplicationConfig.java` via `LocalContainerEntityManagerFactoryBean` and `HibernateJpaVendorAdapter`, Hibernate translates JPA operations into SQL for the MySQL database.
 - Example: `entityManager.persist(user)` in `createUser` is handled by Hibernate to generate an `INSERT` statement.

2. Schema Generation:

- **Purpose:** Hibernate automatically generates and updates the database schema based on entity mappings.
- **Implementation:** The `hibernate.hbm2ddl.auto = update` property in `ApplicationConfig.java` instructs Hibernate to create or update tables (`users`, `user_profiles`, `posts`, `roles`, `user_roles`) based on entity annotations.
 - Example: The `@Table` and `@Column` annotations define table and column names, and Hibernate creates foreign key constraints (e.g., `user_id` in `user_profiles`, `author_id` in `posts`).

3. SQL Dialect:

- **Purpose:** Hibernate uses a database-specific dialect to generate SQL compatible with MySQL 5.7.
- **Implementation:** The `hibernate.dialect = org.hibernate.dialect.MySQL5Dialect` property in `ApplicationConfig.java` ensures Hibernate generates MySQL-compatible SQL, handling data types like `DATETIME` for `LocalDateTime` fields (`created_at`).

4. Lazy and Eager Loading:

- **Purpose:** Hibernate manages fetch strategies for relationships to optimize performance.



- **Implementation:** Annotations like FetchType.LAZY in Post.java (@ManyToOne) and User.java (@OneToOne, @OneToMany) defer loading of related entities until accessed, reducing unnecessary queries.
 - Example: In findPostsByUser, Hibernate executes a JPQL query with a JOIN FETCH to eagerly load posts with their authors when needed.

5. Cascading Operations:

- **Purpose:** Hibernate supports cascading operations to propagate persistence actions to related entities.
- **Implementation:**
 - CascadeType.ALL in User.java for UserProfile and Post ensures that saving or deleting a user cascades to its profile and posts.
 - CascadeType.PERSIST and MERGE for Role in the ManyToMany mapping ensure role assignments are persisted or updated but not deleted when a user is removed.
 - Example: In UserService.createUserWithProfile, persisting a User also persists its UserProfile due to cascading.

6. Query Optimization:

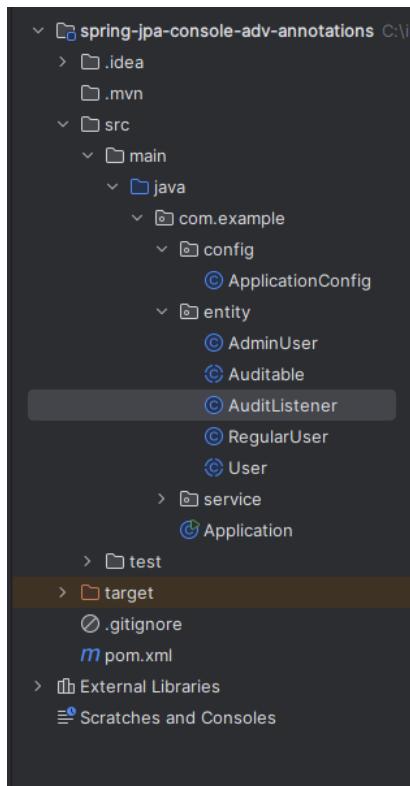
- **Purpose:** Hibernate optimizes JPQL queries and provides additional features like caching and batch processing.
- **Implementation:** The hibernate.show_sql = true and hibernate.format_sql = true properties in ApplicationConfig.java allow developers to inspect generated SQL, ensuring efficient queries.
 - Example: The findAllUsers method uses a simple JPQL query (SELECT u FROM User u), which Hibernate translates into an optimized SELECT statement.

7. Connection Management:

- **Purpose:** Hibernate integrates with HikariCP (configured in ApplicationConfig.java) to manage database connections efficiently.
- **Implementation:** The DataSource bean provides a connection pool, and Hibernate uses it to execute SQL statements, ensuring performance and scalability.



Simplified MySQL console-based CRUD application using Spring JPA to demonstrate the specified JPA annotations: @NamedQuery, @NamedQueries, @Inheritance, @DiscriminatorColumn, @EntityListeners, @PrePersist, @PostPersist, and @Version. The application will extend the concepts from the provided Spring JPA CRUD application, focusing on a minimal set of entities to showcase these annotations while maintaining Java 1.8 compatibility.



Project Overview

- **Purpose:** Demonstrate JPA annotations (@NamedQuery, @NamedQueries, @Inheritance, @DiscriminatorColumn, @EntityListeners, @PrePersist, @PostPersist, @Version) in a console-based CRUD application.
- **Entities:**
 - User: An abstract base entity with inheritance (for RegularUser and AdminUser subclasses).
 - Auditable: A base class for auditing (creation and update timestamps).
 - RegularUser and AdminUser: Subclasses of User to demonstrate inheritance.
- **Database:** MySQL (jpa_example_db) with a single table for users (using SINGLE_TABLE inheritance strategy).
- **Features:**
 - CRUD operations for users via a console interface.
 - Named queries to find all active users and users by email.
 - Auditing of creation and update timestamps using an entity listener.
 - Optimistic locking with version control.
- **Dependencies:** Spring 5, Hibernate 5, MySQL Connector/J 5.1, HikariCP (Java 1.8 compatible).



File Structure and Descriptions

1. **Auditable.java:** Abstract base class with auditing fields (createdAt, updatedAt).

```
package com.example.entity;

import jakarta.persistence.EntityListeners;
import jakarta.persistence.MappedSuperclass;
import jakarta.persistence.Temporal;
import jakarta.persistence.TemporalType;
import java.util.Date;

@MappedSuperclass
@EntityListeners(AuditListener.class)
public abstract class Auditable {

    @Temporal(TemporalType.TIMESTAMP)
    private Date createdAt;

    @Temporal(TemporalType.TIMESTAMP)
    private Date updatedAt;

    public Date getCreatedAt() {
        return createdAt;
    }

    public void setCreatedAt(Date createdAt) {
        this.createdAt = createdAt;
    }

    public Date getUpdatedAt() {
        return updatedAt;
    }

    public void setUpdatedAt(Date updatedAt) {
        this.updatedAt = updatedAt;
    }
}
```

2. **AuditListener.java:** Entity listener for handling @PrePersist and @PostPersist to set timestamps.

```
3. package com.example.entity;

import jakarta.persistence.PrePersist;
import jakarta.persistence.PostPersist;
import java.util.Date;

public class AuditListener {
    @PrePersist
    public void setCreatedAt(Auditable auditable) {
        auditable.setCreatedAt(new Date());
        auditable.setUpdatedAt(new Date());
    }
}
```



```
    }

    @PostPersist
    public void setUpdatedAt(Auditable auditable) {
        auditable.setUpdatedAt(new Date());
    }
}
```

4. **User.java:** Abstract base entity with @Inheritance, @DiscriminatorColumn, @NamedQueries, and @Version.

```
package com.example.entity;

import jakarta.persistence.*;
import java.util.Objects;

@Entity
@Table(name = "users")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "user_type", discriminatorType = DiscriminatorType.STRING)
@NamedQueries({
    @NamedQuery(name = "User.findAllActive", query = "SELECT u FROM User u WHERE u.isActive = true"),
    @NamedQuery(name = "User.findByEmail", query = "SELECT u FROM User u WHERE u.email = :email")
})
public abstract class User extends Auditable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true, nullable = false)
    private String username;

    @Column(nullable = false)
    private String email;

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

    @Version
    private int version;

    // Constructors
    protected User() {
    }

    public User(String username, String email, boolean isActive) {
```



```
        this.username = username;
        this.email = email;
        this.isActive = isActive;
    }

    // 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 getEmail() {
        return email;
    }

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

    public boolean isActive() {
        return isActive;
    }

    public void setActive(boolean active) {
        isActive = active;
    }

    public int getVersion() {
        return version;
    }

    public void setVersion(int version) {
        this.version = version;
    }

    @Override
```



```
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof User)) return false;
    User user = (User) o;
    return Objects.equals(id, user.id);
}

@Override
public int hashCode() {
    return Objects.hash(id);
}

@Override
public String toString() {
    return "User{" +
        "id=" + id +
        ", username='" + username + '\'' +
        ", email='" + email + '\'' +
        ", isActive=" + isActive +
        ", version=" + version +
        '}';
}
}
```

5. **RegularUser.java:** Subclass of User for regular users.

```
package com.example.entity;

import jakarta.persistence.DiscriminatorValue;
import jakarta.persistence.Entity;

@Entity
@DiscriminatorValue("REGULAR")
public class RegularUser extends User {
    public RegularUser() {}

    public RegularUser(String username, String email, boolean isActive) {
        super(username, email, isActive);
    }
}
```

6. **AdminUser.java:** Subclass of User for admin users.

```
package com.example.entity;

import jakarta.persistence.DiscriminatorValue;
```



```
import jakarta.persistence.Entity;  
  
@Entity  
@DiscriminatorValue("ADMIN")  
public class AdminUser extends User {  
    public AdminUser() {}  
  
    public AdminUser(String username, String email, boolean isActive) {  
        super(username, email, isActive);  
    }  
}
```

7. **UserService.java:** Service class for CRUD operations using @NamedQuery.

```
package com.example.service;  
  
import com.example.entity.User;  
import jakarta.persistence.EntityManager;  
import jakarta.persistence.PersistenceContext;  
import jakarta.persistence.TypedQuery;  
import org.springframework.stereotype.Service;  
import org.springframework.transaction.annotation.Transactional;  
  
import java.util.List;  
  
@Service  
@Transactional  
public class UserService {  
  
    @PersistenceContext  
    private EntityManager entityManager;  
  
    public User createUser(User user) {  
        entityManager.persist(user);  
        return user;  
    }  
  
    @Transactional(readOnly = true)  
    public User findUserById(Long id) {  
        return entityManager.find(User.class, id);  
    }  
  
    @Transactional(readOnly = true)  
    public List<User> findAllActiveUsers() {  
        TypedQuery<User> query = entityManager.createNamedQuery("User.findAllActive",  
User.class);  
    }
```



```
        return query.getResultList();
    }

    @Transactional(readOnly = true)
    public User findUserByEmail(String email) {
        TypedQuery<User> query = entityManager.createNamedQuery("User.findByEmail",
User.class);
        query.setParameter("email", email);
        List<User> users = query.getResultList();
        return users.isEmpty() ? null : users.get(0);
    }

    public void updateUser(Long id, String username, String email, boolean isActive) {
        User user = entityManager.find(User.class, id);
        if (user != null) {
            user.setUsername(username);
            user.setEmail(email);
            user.setActive(isActive);
            entityManager.merge(user);
        }
    }

    public void deleteUser(Long id) {
        User user = entityManager.find(User.class, id);
        if (user != null) {
            entityManager.remove(user);
        }
    }
}
```

8. **ApplicationConfig.java:** Spring configuration for JPA, Hibernate, and HikariCP.

```
package com.example.config;

import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;
import java.util.Properties;
```



```
@Configuration
@ComponentScan(basePackages = "com.example")
@EnableTransactionManagement
public class ApplicationConfig {

    @Bean
    public DataSource dataSource() {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:mysql://localhost:3306/jpa_adv_example_db");
        config.setUsername("root");
        config.setPassword("Archer@12345");
        config.setDriverClassName("com.mysql.cj.jdbc.Driver");
        config.setMaximumPoolSize(10);
        config.setMinimumIdle(5);
        config.setConnectionTimeout(30000);
        config.setIdleTimeout(600000);
        config.setMaxLifetime(1800000);
        return new HikariDataSource(config);
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean emf = new
        LocalContainerEntityManagerFactoryBean();
        emf.setDataSource(dataSource());
        emf.setPackagesToScan("com.example.entity");
        emf.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
        emf.setJpaProperties(jpaProperties());
        return emf;
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        JpaTransactionManager txManager = new JpaTransactionManager();
        txManager.setEntityManagerFactory(entityManagerFactory().getObject());
        return txManager;
    }

    private Properties jpaProperties() {
        Properties properties = new Properties();
        properties.put("hibernate.dialect", "org.hibernate.dialect.MySQL8Dialect");
        properties.put("hibernate.hbm2ddl.auto", "update");
        properties.put("hibernate.show_sql", "true");
        properties.put("hibernate.format_sql", "true");
        return properties;
    }
}
```



```
    }  
}
```

9. Application.java: Console interface for CRUD operations.

```
package com.example;  
  
import com.example.config.ApplicationConfig;  
import com.example.entity.AdminUser;  
import com.example.entity.RegularUser;  
import com.example.entity.User;  
import com.example.service.UserService;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
import java.util.List;  
import java.util.Scanner;  
  
public class Application {  
    public static void main(String[] args) {  
        System.out.println("== Spring JPA CRUD Demo with Annotations ==\n");  
  
        AnnotationConfigApplicationContext context =  
            new AnnotationConfigApplicationContext(ApplicationConfig.class);  
        UserService userService = context.getBean(UserService.class);  
        Scanner scanner = new Scanner(System.in);  
  
        try {  
            while (true) {  
                System.out.println("\nUser Management Menu:");  
                System.out.println("1. Create Regular User");  
                System.out.println("2. Create Admin User");  
                System.out.println("3. Read User by ID");  
                System.out.println("4. Find Active Users");  
                System.out.println("5. Find User by Email");  
                System.out.println("6. Update User");  
                System.out.println("7. Delete User");  
                System.out.println("8. Exit");  
                System.out.print("Enter your choice: ");  
  
                int choice = scanner.nextInt();  
                scanner.nextLine(); // Consume newline  
  
                switch (choice) {  
                    case 1:  
                        System.out.print("Enter username: ");  
                        String regUsername = scanner.nextLine();  
                        System.out.print("Enter email: ");
```



```
String regEmail = scanner.nextLine();
System.out.print("Is active (true/false): ");
boolean regActive = scanner.nextBoolean();
scanner.nextLine(); // Consume newline
RegularUser regUser = new RegularUser(regUsername, regEmail, regActive);
userService.createUser(regUser);
System.out.println("Created: " + regUser);
break;
case 2:
    System.out.print("Enter username: ");
    String adminUsername = scanner.nextLine();
    System.out.print("Enter email: ");
    String adminEmail = scanner.nextLine();
    System.out.print("Is active (true/false): ");
    boolean adminActive = scanner.nextBoolean();
    scanner.nextLine(); // Consume newline
    AdminUser adminUser = new AdminUser(adminUsername, adminEmail,
adminActive);
    userService.createUser(adminUser);
    System.out.println("Created: " + adminUser);
    break;
case 3:
    System.out.print("Enter user ID: ");
    Long id = scanner.nextLong();
    scanner.nextLine(); // Consume newline
    User user = userService.findUserById(id);
    if (user != null) {
        System.out.println("Found: " + user);
    } else {
        System.out.println("User not found");
    }
    break;
case 4:
    List<User> activeUsers = userService.findAllActiveUsers();
    System.out.println("Active Users:");
    for (User u : activeUsers) {
        System.out.println(" " + u);
    }
    break;
case 5:
    System.out.print("Enter email: ");
    String email = scanner.nextLine();
    User userByEmail = userService.findUserByEmail(email);
    if (userByEmail != null) {
        System.out.println("Found: " + userByEmail);
    } else {
        System.out.println("User not found");
```



```
        }
        break;
    case 6:
        System.out.print("Enter user ID: ");
        Long updateId = scanner.nextLong();
        scanner.nextLine(); // Consume newline
        System.out.print("Enter new username: ");
        String newUsername = scanner.nextLine();
        System.out.print("Enter new email: ");
        String newEmail = scanner.nextLine();
        System.out.print("Is active (true/false): ");
        boolean newActive = scanner.nextBoolean();
        scanner.nextLine(); // Consume newline
        userService.updateUser(updateId, newUsername, newEmail, newActive);
        System.out.println("User updated");
        break;
    case 7:
        System.out.print("Enter user ID: ");
        Long deleteId = scanner.nextLong();
        scanner.nextLine(); // Consume newline
        userService.deleteUser(deleteId);
        System.out.println("User deleted");
        break;
    case 8:
        System.out.println("Exiting...");
        scanner.close();
        context.close();
        return;
    default:
        System.out.println("Invalid choice. Please try again.");
    }
}
} catch (Exception e) {
    System.err.println("Error occurred: " + e.getMessage());
    e.printStackTrace();
} finally {
    scanner.close();
    context.close();
}
}
```

Database Schema

- **Table:** users
 - **Columns:**
 - id (BIGINT, PRIMARY KEY, AUTO_INCREMENT)



- user_type (VARCHAR, discriminator for inheritance, e.g., 'REGULAR', 'ADMIN')
 - username (VARCHAR, UNIQUE, NOT NULL)
 - email (VARCHAR, NOT NULL)
 - is_active (BOOLEAN, NOT NULL)
 - created_at (DATETIME, from Auditable)
 - updated_at (DATETIME, from Auditable)
 - version (INTEGER, for optimistic locking)
- **Description:** Uses SINGLE_TABLE inheritance, with user_type distinguishing between RegularUser and AdminUser.

Demonstration of Annotations:

1. **@NamedQuery, @NamedQueries:**
 - **Usage:** In User.java, @NamedQueries defines User.findAllActive and User.findByEmail.
 - **Demonstrated:** UserService.findAllActiveUsers and findUserByEmail use these named queries to retrieve active users and users by email, accessible via menu options 4 and 5 in Application.java.
2. **@Inheritance:**
 - **Usage:** In User.java, @Inheritance(strategy = InheritanceType.SINGLE_TABLE) specifies that RegularUser and AdminUser share the users table.
 - **Demonstrated:** Menu options 1 and 2 in Application.java create RegularUser and AdminUser instances, stored in the users table with different user_type values.
3. **@DiscriminatorColumn:**
 - **Usage:** In User.java, @DiscriminatorColumn(name = "user_type") defines the user_type column to distinguish REGULAR and ADMIN users.
 - **Demonstrated:** Creating users via menu options 1 and 2 sets user_type to REGULAR or ADMIN in the users table.
4. **@EntityListeners:**
 - **Usage:** In Auditable.java, @EntityListeners(AuditListener.class) links the AuditListener to set timestamps.
 - **Demonstrated:** Creating or updating users (menu options 1, 2, 6) triggers AuditListener to set created_at and updated_at.
5. **@PrePersist, @PostPersist:**
 - **Usage:** In AuditListener.java, these annotations set createdAt and updatedAt before and after persistence.
 - **Demonstrated:** Timestamps are automatically set when users are created or updated, visible in the users table (created_at, updated_at).
6. **@Version:**
 - **Usage:** In User.java, @Version on the version field enables optimistic locking.
 - **Demonstrated:** Updating a user (menu option 6) increments the version column, and concurrent updates would throw an OptimisticLockException if conflicts occur.



Can Normal SQL Queries Be Used in @NamedQuery?

Short Answer: No, @NamedQuery does not support native SQL queries directly. The @NamedQuery annotation is designed specifically for **JPQL (Java Persistence Query Language)** queries, which operate on entity objects and their mappings rather than raw database tables. However, JPA provides a separate annotation, @NamedNativeQuery, for defining reusable **native SQL queries**. Both can be used in an application, but they serve different purposes and have distinct syntaxes.

Let's Rewrite the above application as a Web Application:

Lets see, how to configure the DispatcherServlet in detail:

In a **Spring Core** application (typically a Spring MVC application), the DispatcherServlet is the central servlet that handles all HTTP requests and delegates them to appropriate controllers. Configuring the DispatcherServlet can be done either through **XML configuration or Java-based configuration**. Below, I'll explain both approaches clearly and concisely.

1. XML-Based Configuration

This approach uses a web.xml file to configure the DispatcherServlet in a traditional Spring MVC application.

Steps:

1. Add DispatcherServlet to web.xml:

- Define the DispatcherServlet in the WEB-INF/web.xml file.
- Map it to specific URL patterns to handle requests.

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
    version="4.0">

    <!-- Define DispatcherServlet -->
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- Specify the location of Spring configuration file -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring-config.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- Map DispatcherServlet to URL patterns -->
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
```



```
<url-pattern>/</url-pattern> <!-- Handles all requests -->
</servlet-mapping>
</web-app>
```

2. Create Spring Configuration File:

- Create a file (e.g., spring-config.xml) in WEB-INF/ to define beans, component scanning, and other configurations.
- Example spring-config.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/mvc
                           http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!-- Enable component scanning -->
    <context:component-scan base-package="com.example.package"/>

    <!-- Enable MVC annotations -->
    <mvc:annotation-driven/>

    <!-- View resolver for JSPs -->
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

3. Explanation:

- <servlet-class>: Specifies the DispatcherServlet class from Spring MVC.
- <init-param>: Points to the Spring configuration file (spring-config.xml).
- <url-pattern>: Maps the servlet to handle requests (e.g., / for all requests).
- <context:component-scan>: Scans for Spring components (e.g., @Controller, @Service).
- <mvc:annotation-driven>: Enables Spring MVC annotations like @RequestMapping.
- InternalResourceViewResolver: Resolves view names to JSP files (or other view technologies).

2. Java-Based Configuration (Spring Boot or Servlet 3.0+)

In modern Spring applications (especially Spring Boot), you can configure the DispatcherServlet programmatically using Java configuration, avoiding XML.



Steps:**1. Create a Configuration Class:**

- o Create a class annotated with @Configuration and @EnableWebMvc to configure Spring MVC.
- o Extend WebMvcConfigurer to customize MVC settings.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.example.package")
public class WebConfig implements WebMvcConfigurer {

    @Bean
    public InternalResourceViewResolver viewResolver() {
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        return resolver;
    }
}
```

2. Initialize DispatcherServlet:

- o Create a class that extends AbstractAnnotationConfigDispatcherServletInitializer to register the DispatcherServlet programmatically.

```
import
org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class WebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[] {};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] {WebConfig.class}; // Reference the configuration class
    }
}
```



```
}

@Override
protected String[] getServletMappings() {
    return new String[]{"/*"}; // Map to all requests
}
}
```

3. Explanation:

- @EnableWebMvc: Enables Spring MVC features.
- @ComponentScan: Scans for Spring components in the specified package.
- InternalResourceViewResolver: Configures the view resolver for JSPs.
- AbstractAnnotationConfigDispatcherServletInitializer: Registers the DispatcherServlet and maps it to a URL pattern (e.g., /).
- No web.xml is needed, as this is a Servlet 3.0+ approach.

The provided code is a Java class that configures a Spring MVC web application using Spring's Java-based configuration. It extends AbstractAnnotationConfigDispatcherServletInitializer, a base class provided by Spring to simplify the initialization of the DispatcherServlet and the Spring application context in a Servlet 3.0+ container (e.g., Tomcat, Jetty). This class is typically used to replace the traditional web.xml file for configuring a Spring web application. Below, I'll break down the code in detail, explaining each part, its purpose, and how it contributes to the Spring MVC setup.

Code Breakdown

```
import
org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;
```

- **Purpose:** This imports the AbstractAnnotationConfigDispatcherServletInitializer class from the Spring Framework's web.servlet.support package.
- **Role:** This class is a convenience base class for initializing a Spring MVC DispatcherServlet and setting up the Spring application context in a Servlet 3.0+ environment. It leverages Java-based configuration (annotations) instead of XML-based configuration (e.g., web.xml).
- **Servlet 3.0+:** The Servlet 3.0 specification allows programmatic configuration of servlets, filters, and listeners, which this class uses to configure the Spring MVC application.

```
public class WebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer
{
```

- **Purpose:** Defines a class named WebAppInitializer that extends AbstractAnnotationConfigDispatcherServletInitializer.
- **Role:** By extending this class, WebAppInitializer provides the necessary configuration for initializing the Spring MVC DispatcherServlet and the application context. It acts as the entry point for the Spring MVC application, replacing the need for a web.xml file.



- **Why extend?**: The base class provides default implementations for many initialization tasks, and you override specific methods to customize the configuration (e.g., specifying configuration classes and servlet mappings).

Overridden Methods

The class overrides three abstract methods from `AbstractAnnotationConfigDispatcherServletInitializer` to configure the Spring application context and the DispatcherServlet. Let's examine each method in detail:

1. `getRootConfigClasses()`

```
protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[]{};
```

}

- **Purpose**: Specifies the configuration classes for the **root application context**.
- **Details**:
 - The **root application context** is shared across all servlets in the application and typically contains configuration for services, repositories, data sources, and other backend components (e.g., database connections, business logic).
 - This method returns an array of Java configuration classes (annotated with `@Configuration`) that define beans for the root context.
 - In this case, `new Class<?>[]{}` returns an empty array, meaning **no root configuration classes are specified**. This implies that the application either:
 - Does not need a root application context (e.g., all configuration is handled in the servlet context).
 - Or relies on other mechanisms (e.g., auto-configuration or default settings) for the root context.
 - **When to use a root context?**: If your application has components like database configurations, security settings, or other global services, you would return their configuration classes here (e.g., `return new Class<?>[]{{AppConfig.class}};`).

2. `getServletConfigClasses()`

```
@Override
protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[]{{WebConfig.class}};
}
```

- **Purpose**: Specifies the configuration classes for the **dispatcher application context**.
- **Details**:
 - The **dispatcher application context** is specific to the DispatcherServlet and typically contains web-related components, such as controllers, view resolvers, and handler mappings.
 - This method returns an array of Java configuration classes that define beans for the DispatcherServlet's context.
 - Here, `WebConfig.class` is specified, indicating that the `WebConfig` class (a `@Configuration`-annotated class) contains the configuration for web-related components (e.g., `@Controller`, `@Bean` definitions for view resolvers, etc.).
 - **Example WebConfig class** (not shown in the provided code but implied):



```
@Configuration  
@EnableWebMvc  
@ComponentScan(basePackages = "com.example.controllers")  
public class WebConfig {  
    @Bean  
    public ViewResolver viewResolver() {  
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();  
        resolver.setPrefix("/WEB-INF/views/");  
        resolver.setSuffix(".jsp");  
        return resolver;  
    }  
}
```

- This hypothetical WebConfig class might enable Spring MVC (`@EnableWebMvc`), scan for controllers, and configure a view resolver for rendering JSP pages.

- **Key Point:** The WebConfig class is responsible for setting up the MVC-specific configuration, while the root context (if used) handles non-web components.

3. `getServletMappings()`

```
@Override  
protected String[] getServletMappings() {  
    return new String[]{"/"};  
}
```

- **Purpose:** Defines the URL patterns that the DispatcherServlet will handle.
- **Details:**
 - This method specifies which HTTP requests should be routed to the Spring MVC DispatcherServlet.
 - Returning `new String[]{"/"}` means the DispatcherServlet will handle **all requests** to the application (root URL and its sub-paths).
 - **Implications:**
 - The `"/"` mapping makes the DispatcherServlet the default servlet, handling all incoming requests unless overridden by other servlets or filters.
 - This is common in Spring MVC applications, as it allows the DispatcherServlet to process requests for controllers, static resources (if configured), and other endpoints.
 - If you want to limit the servlet to specific paths (e.g., `/api/*`), you could return `new String[]{"/api/*"}`.
 - **Static Resources:** If the DispatcherServlet maps to `"/"`, you may need to configure static resource handling (e.g., for CSS, JS, or images) in the WebConfig class using WebMvcConfigurer's `addResourceHandlers` method to avoid the servlet intercepting those requests.



Configuration in case of Thymeleaf:

Key Changes When Switching from JSP to Thymeleaf

1. Update Dependencies

- **JSP Dependency Removal:**

- Remove JSP-related dependencies (e.g., javax.servlet.jsp or jakarta.servlet.jsp) and any JSP-specific libraries (e.g., JSTL) from your build file.

- **Add Thymeleaf Dependency:**

- Add the Thymeleaf dependency for Spring integration.
- **Maven Example (Spring Core):**

```
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf-spring6</artifactId>
    <version>3.1.2.RELEASE</version> <!-- Use the latest compatible version -->
</dependency>
    ○ Spring Boot:
        ▪ Use the spring-boot-starter-thymeleaf dependency, which includes Thymeleaf and its Spring integration.
        ▪ Maven Example:
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
    <version>3.2.5</version> <!-- Use the latest version -->
</dependency>
    ○ Note: Ensure the Thymeleaf version is compatible with Spring Framework 6 (e.g., thymeleaf-spring6 for Spring 6.x).
```

2. Replace View Resolver Configuration

- **JSP View Resolver:**

- JSP uses InternalResourceViewResolver to resolve .jsp files located in a specific directory (e.g., /WEB-INF/views/).
- Example (JSP):

```
@Bean
public InternalResourceViewResolver viewResolver() {
    InternalResourceViewResolver resolver = new InternalResourceViewResolver();
    resolver.setPrefix("/WEB-INF/views/");
    resolver.setSuffix(".jsp");
    return resolver;
}
```

- **Thymeleaf View Resolver:**

- Replace InternalResourceViewResolver with ThymeleafViewResolver and configure a SpringTemplateEngine for Thymeleaf.



- **Java Configuration (Spring Core):**

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.thymeleaf.spring6.SpringTemplateEngine;
import org.thymeleaf.spring6.templateresolver.SpringResourceTemplateResolver;
import org.thymeleaf.spring6.view.ThymeleafViewResolver;
import org.thymeleaf.templatemode.TemplateMode;

@Configuration
public class WebConfig {

    @Bean
    public SpringResourceTemplateResolver templateResolver() {
        SpringResourceTemplateResolver resolver = new SpringResourceTemplateResolver();
        resolver.setPrefix("/WEB-INF/templates/"); // Thymeleaf templates location
        resolver.setSuffix(".html"); // Thymeleaf uses .html files
        resolver.setTemplateMode(TemplateMode.HTML); // HTML5 mode
        resolver.setCacheable(true); // Enable caching for production
        return resolver;
    }

    @Bean
    public SpringTemplateEngine templateEngine() {
        SpringTemplateEngine engine = new SpringTemplateEngine();
        engine.setTemplateResolver(templateResolver());
        return engine;
    }

    @Bean
    public ThymeleafViewResolver viewResolver() {
        ThymeleafViewResolver resolver = new ThymeleafViewResolver();
        resolver.setTemplateEngine(templateEngine());
        resolver.setCharacterEncoding("UTF-8");
        return resolver;
    }
}
```

- **XML Configuration (Spring Core):**

```
<bean id="templateResolver"
class="org.thymeleaf.spring6.templateresolver.SpringResourceTemplateResolver">
<property name="prefix" value="/WEB-INF/templates/">
<property name="suffix" value=".html"/>
<property name="templateMode" value="HTML"/>
<property name="cacheable" value="true"/>
</bean>
```



```
<bean id="templateEngine" class="org.thymeleaf.spring6.SpringTemplateEngine">
    <property name="templateResolver" ref="templateResolver"/>
</bean>
```

```
<bean class="org.thymeleaf.spring6.view.ThymeleafViewResolver">
    <property name="templateEngine" ref="templateEngine"/>
    <property name="characterEncoding" value="UTF-8"/>
</bean>
```

3. Update Template Files

- **JSP Files:**

- JSP files typically use .jsp extension, reside in /WEB-INF/views/, and use JSP tags (e.g., JSTL <c:forEach>, <c:if>) or scriptlets.
- Example JSP:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<body>
    <h1>Hello, ${name}!</h1>
    <c:forEach var="item" items="${items}">
        <p>${item}</p>
    </c:forEach>
</body>
</html>
```

- **Thymeleaf Templates:**

- Thymeleaf uses .html files (by default) and is located in /WEB-INF/templates/ (Spring Core) or src/main/resources/templates/ (Spring Boot).
- Thymeleaf uses its own syntax with attributes like th:text, th:each, and th:if for dynamic content.
- Example Thymeleaf Template:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Hello Page</title>
</head>
<body>
    <h1 th:text="'Hello, ' + ${name} + '!"'>Hello</h1>
    <div th:each="item : ${items}">
        <p th:text="${item}">Item</p>
    </div>
</body>
</html>
```

- **Key Differences:**

- Thymeleaf templates are valid HTML, allowing them to be previewed in browsers without a server.



- Replace JSTL tags with Thymeleaf attributes (e.g., <c:forEach> → th:each, <c:if> → th:if).
- Use th:text or th:utext for displaying model attributes instead of \${} expressions directly.

7. Handle Static Resources

- **JSP:**
 - Static resources (e.g., CSS, JavaScript) are often served from /WEB-INF/ or other directories and accessed directly in JSPs.
- **Thymeleaf:**
 - Static resources are typically placed in src/main/resources/static/ (Spring Boot) or /WEB-INF/static/ (Spring Core) and referenced using Thymeleaf's @{} syntax for context-relative URLs.
 - Example:

```
<link rel="stylesheet" th:href="@{/css/style.css}" />
<script th:src="@{/js/script.js}"></script>
```

Spring Core Configuration:

- Ensure static resources are served by adding a resource handler:

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/static/**").addResourceLocations("/WEB-INF/static/");
    }
}
```

XML Equivalent:

```
<mvc:resources mapping="/static/**" location="/WEB-INF/static/" />
```

Spring Boot:

Static resources in src/main/resources/static/ are served automatically under /.

Using React as a UI:

When using **React** for the UI in a Spring Core or Spring Boot application, the configuration of the DispatcherServlet and the overall application architecture changes significantly compared to using server-side view technologies like JSP or Thymeleaf. React is a client-side JavaScript library for building user interfaces, typically served as a single-page application (SPA). This shifts the application to a **frontend-backend separation** model, where Spring serves as a REST API backend, and React handles the UI rendering in the browser. Below, I outline the key changes to the DispatcherServlet configuration and related setup when integrating React for the UI, considering Spring Framework 6.x (and Spring Boot 3.x if applicable) in the context of your previous questions.



Key Changes When Using React for UI

1. Architectural Shift: REST API Backend

- **JSP/Thymeleaf:** The DispatcherServlet maps requests to controllers that return view names (e.g., hello resolves to hello.jsp or hello.html), and the server renders the UI using a view resolver.
- **React:** The DispatcherServlet primarily handles REST API endpoints (returning JSON data via @RestController), and the React frontend, served as static HTML, CSS, and JavaScript, consumes these APIs. The React app is typically a single HTML file (e.g., index.html) that loads the JavaScript bundle for client-side rendering.

Using SQL functions, views, triggers, and stored procedures in Spring-JPA

The **User Management Web Application** is a Spring MVC-based web application designed to demonstrate the integration of **Spring JPA** (Java Persistence API) with **MySQL** as the database and **JSP** (JavaServer Pages) as the user interface. Built using **Spring Framework 6.x** and compatible with **Jakarta EE 9+**, this application provides a robust platform for managing user data, showcasing advanced database operations, and leveraging modern Java technologies.

The application allows users to perform CRUD (Create, Read, Update, Delete) operations on user records, utilizing MySQL's advanced features such as **SQL functions, views, triggers, and stored procedures**. These database constructs are seamlessly integrated with Spring JPA, using EntityManager for direct database interactions without relying on Spring Data JPA. The frontend, built with JSP, provides a simple and intuitive interface for listing users, adding new users, and displaying computed data like full names using database functions and stored procedures.

Key Features

- **Database Integration:** Connects to a MySQL database with a user table and a user_summary view, leveraging triggers for automatic timestamping and custom functions/stored procedures for data processing.
- **Spring JPA:** Uses pure JPA with Hibernate as the provider, implementing custom repositories with EntityManager for flexible query execution.
- **Spring MVC:** Employs the DispatcherServlet to handle HTTP requests, mapping them to controllers that render JSP views.
- **JSP UI:** Provides a user-friendly interface for interacting with user data, including forms for data entry and tables for data display.
- **Jakarta EE 9+ Compliance:** Ensures compatibility with modern Java EE standards, using jakarta.* APIs and running on Java 17+.

This project serves as a practical example for developers looking to integrate Spring JPA with MySQL in a web application, demonstrating best practices for handling complex database operations while maintaining a clean MVC architecture. It is deployable on Jakarta EE 9+-compatible servers like Tomcat 10.1+ and is designed for scalability and maintainability.



Advanced Database Concepts in Spring JPA Application

Concept	Short Description
Functions	Reusable SQL code blocks that perform computations and return a single value, used for custom logic like concatenating names.
Views	Virtual tables based on stored queries, providing filtered or summarized data (e.g., active users) as read-only entities.
Triggers	Automated procedures executed on specific table events (e.g., INSERT), ensuring data consistency like setting timestamps.
Stored Procedures	Parameterized SQL routines stored in the database, executing complex logic (e.g., formatting user details) efficiently.

- Purpose:** This projection highlights how these concepts enhance the application by offloading database-specific operations, improving performance, and maintaining data integrity, while Spring JPA manages object-relational mapping and business logic.
- Context:** Applied in a user management system with endpoints for listing active users (/users), adding users (/users/add), and retrieving full names (/users/{id}/fullname) or stored procedure results (/users/{id}/stored).

List of Files with Significance

schema.sql (located in src/main/resources/schema.sql): Defines the database structure including the user table, get_full_name function for concatenating names, get_user_by_id stored procedure for retrieving formatted user data, user_summary view for active users, and before_user_insert trigger for setting created_at, serving as the foundation for all database operations.

```
-- Create the database
```

```
CREATE DATABASE IF NOT EXISTS spring_db;
USE spring_db;
```

```
-- Create the user table
```

```
CREATE TABLE user (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    email VARCHAR(100),
    created_at DATETIME,
    active TINYINT DEFAULT 1
);
```

```
-- Create the get_full_name function
```

```
DELIMITER //
CREATE FUNCTION get_full_name(first_name VARCHAR(50), last_name VARCHAR(50))
RETURNS VARCHAR(100)
DETERMINISTIC
BEGIN
```



```
RETURN CONCAT(first_name, ' ', last_name);
END //
DELIMITER ;

-- Create the get_user_by_id stored procedure
DELIMITER //
CREATE PROCEDURE get_user_by_id(IN userId INT, OUT userName VARCHAR(150))
BEGIN
    SELECT CONCAT(first_name, ' ', last_name, ' - ', email) INTO userName
    FROM user
    WHERE id = userId;
END //
DELIMITER ;

-- Create the user_summary view
CREATE OR REPLACE VIEW user_summary AS
SELECT id, first_name, last_name, email, active
FROM user
WHERE active = 1;

-- Create the before_user_insert trigger
DELIMITER //
CREATE TRIGGER before_user_insert
BEFORE INSERT ON user
FOR EACH ROW
BEGIN
    SET NEW.created_at = NOW();
END //
DELIMITER ;
```

User.java (located in `src/main/java/com/example/model/`): Acts as the JPA entity mapping the user table, includes `@NamedStoredProcedureQuery` to integrate the `get_user_by_id` stored procedure, and supports CRUD operations and data persistence.

```
package com.example.model;

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

@Entity
@Table(name = "user")
@NamedStoredProcedureQuery(
    name = "getUserById",
    procedureName = "get_user_by_id",
    parameters = {
        @StoredProcedureParameter(mode = ParameterMode.IN, name = "userId", type =
Integer.class),
```



```

        @StoredProcedureParameter(mode = ParameterMode.OUT, name = "userName",
type = String.class)
    }
)
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String first_name;
    private String last_name;
    private String email;
    @Column(name = "created_at", updatable = false)
    private LocalDateTime createdAt;
    @Column(name = "active")
    private Boolean active = true;

    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getFirstName() { return first_name; }
    public void setFirstName(String firstName) { this.first_name = firstName; }
    public String getLastname() { return last_name; }
    public void setLastName(String lastName) { this.last_name = lastName; }
    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
    public LocalDateTime getCreatedAt() { return createdAt; }
    public void setCreatedAt(LocalDateTime createdAt) { this.createdAt = createdAt; }
    public Boolean getActive() { return active; }
    public void setActive(Boolean active) { this.active = active; }
}

```

UserSummary.java (located in src/main/java/com/example/model/): Functions as a read-only JPA entity with `@Immutable` mapping the `user_summary` view, enabling efficient querying of active users without modification capabilities.

```

package com.example.model;

import jakarta.persistence.*;
import org.hibernate.annotations.Immutable;

@Entity
@Immutable
@Table(name = "user_summary")
public class UserSummary {
    @Id
    private Long id;
    @Column(name = "first_name")

```



```
private String firstName;
@Column(name = "last_name")
private String lastName;
private String email;
@Column(name = "active")
private Boolean active;

public Long getId() { return id; }
public void setId(Long id) { this.id = id; }
public String getFirstName() { return firstName; }
public void setFirstName(String firstName) { this.firstName = firstName; }
public String getLastName() { return lastName; }
public void setLastName(String lastName) { this.lastName = lastName; }
public String getEmail() { return email; }
public void setEmail(String email) { this.email = email; }
public Boolean getActive() { return active; }
public void setActive(Boolean active) { this.active = active; }
}
```

UserService.java (located in src/main/java/com/example/service/): Serves as the business logic layer, orchestrating calls to native queries (e.g., getFullName using the get_full_name function), stored procedures (e.g., callStoredProcedure), and repository methods, with `@Transactional` ensuring data consistency.

```
package com.example.service;

import com.example.model.User;
import com.example.model.UserSummary;
import com.example.repository.UserRepository;
import com.example.repository.UserSummaryRepository;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.persistence.StoredProcedureQuery;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service
@Transactional
public class UserService {
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private UserSummaryRepository userSummaryRepository;
```



```
@PersistenceContext  
private EntityManager entityManager;  
  
public User saveUser(User user) {  
    return userRepository.save(user); // Trigger sets createdAt  
}  
  
public String getFullName(Long id) {  
    return userRepository.getFullName(id); // Calls SQL function  
}  
  
public List<UserSummary> getUserSummariesByEmail(String email) {  
    return userSummaryRepository.findByEmail(email); // Queries view  
}  
  
public String call.StoredProcedure(Long id) {  
    StoredProcedureQuery query = entityManager  
        .createNamedStoredProcedureQuery("getUserById")  
        .setParameter("userId", id.intValue());  
    query.execute();  
    return (String) query.getOutputParameterValue("userName");  
}  
}
```

UserRepository.java (located in src/main/java/com/example/repository/): Handles CRUD operations and executes native SQL queries (e.g., getFullName) for the User entity, bridging JPA with database-specific logic like the get_full_name function.

```
package com.example.repository;  
  
import com.example.model.User;  
import jakarta.persistence.EntityManager;  
import jakarta.persistence.PersistenceContext;  
import jakarta.persistence.Query;  
import org.springframework.stereotype.Repository;  
import org.springframework.transaction.annotation.Transactional;  
  
@Repository  
@Transactional  
public class UserRepository {  
    @PersistenceContext  
    private EntityManager entityManager;  
  
    public User save(User user) {  
        if (user.getId() == null) {  
            entityManager.persist(user);  
        } else {  
            entityManager.merge(user);  
        }  
    }  
}
```



```
        return user;
    } else {
        return entityManager.merge(user);
    }
}

public User findById(Long id) {
    return entityManager.find(User.class, id);
}

public String getFullName(Long id) {
    Query query = entityManager.createNativeQuery(
        "SELECT get_full_name(u.first_name, u.last_name) AS fullName FROM user u
WHERE u.id = :id"
    );
    query.setParameter("id", id);
    return (String) query.getSingleResult();
}
}
```

UserSummaryRepository.java (located in src/main/java/com/example/repository/):
Provides JPQL queries (e.g., findByEmail) for the UserSummary entity, leveraging the user_summary view to fetch active user data efficiently.

```
package com.example.repository;

import com.example.model.UserSummary;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.persistence.TypedQuery;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Repository
@Transactional
public class UserSummaryRepository {
    @PersistenceContext
    private EntityManager entityManager;

    public List<UserSummary> findByEmail(String email) {
        String jpql = "SELECT u FROM UserSummary u";
        if (email != null) {
            jpql += " WHERE u.email = :email";
        }
    }
}
```



```
TypedQuery<UserSummary> query = entityManager.createQuery(jpql,
UserSummary.class);
if (email != null) {
    query.setParameter("email", email);
}
List<UserSummary> result = query.getResultList();
System.out.println("Query: " + jpql + ", Result size: " + result.size());
return result;
}
}
```

UserController.java (located in src/main/java/com/example/controller/): Exposes REST-like endpoints (/users, /users/add, /users/{id}/fullname, /users/{id}/stored), binds form data from userForm.jsp, and renders responses via JSP views, acting as the web interface layer.

```
package com.example.controller;

import com.example.model.User;
import com.example.model.UserSummary;
import com.example.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@Controller
@RequestMapping("/users")
public class UserController {
    @Autowired
    private UserService userService;

    @GetMapping
    public String listUsers(Model model) {
        model.addAttribute("users", userService.getUserSummariesByEmail(null));

        List<UserSummary> list = userService.getUserSummariesByEmail(null);
        for(UserSummary u : list)
            System.out.println("\n Data: "+u);
        return "userList";
    }

    @GetMapping("/add")
    public String showAddForm(Model model) {
        model.addAttribute("user", new User());
```



```
        return "userForm";
    }

    @PostMapping("/save")
    public String saveUser(@ModelAttribute User user) {
        userService.saveUser(user);
        return "redirect:/users";
    }

    @GetMapping("/{id}/fullname")
    public String getFullName(@PathVariable("id") Long id, Model model) {
        model.addAttribute("fullName", userService.getFullName(id));
        return "userFullName";
    }

    @GetMapping("/{id}/stored")
    public String callStoredProcedure(@PathVariable("id") Long id, Model model) {
        model.addAttribute("fullName", userService.callStoredProcedure(id));
        return "userFullName";
    }
}
```

userList.jsp (located in src/main/webapp/WEB-INF/views/): Displays a list of active users retrieved from the user_summary view, includes links to detail endpoints, and serves as the main user listing page.

```
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
<head>
    <title>User List</title>
</head>
<body>
    <h1>Users</h1>
    <a href="users/add">Add User</a>
    <table border="1">
        <tr>
            <th>ID</th>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Email</th>
            <th>Active</th>
            <th>Actions</th>
        </tr>
        <c:forEach var="user" items="${users}">
            <tr>
                <td>${user.id}</td>
```



```
<td>${user.firstName}</td>
<td>${user.lastName}</td>
<td>${user.email}</td>
<td>${user.active ? 'Active' : 'Inactive'}</td>
<td>
    <a href="users/${user.id}/fullname">View Full Name</a>
    <a href="users/${user.id}/stored">View Stored Proc</a>
</td>
</tr>
</c:forEach>
</table>
</body>
</html>
```

userFullName.jsp (located in src/main/webapp/WEB-INF/views/): Renders the full name (from get_full_name) or stored procedure output (from get_user_by_id), functioning as the detail view for individual users.

```
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
<head>
    <title>User Full Name</title>
</head>
<body>
    <h1>User Details</h1>
    <p><strong>Details:</strong> <c:out value="${fullName}" /></p>
    <a href="${pageContext.request.contextPath}/users">Back to Home</a>
</body>
</html>
```

userForm.jsp (located in src/main/webapp/WEB-INF/views/): Provides a form for adding users, binding to the User entity, and submitting data to the /users/save endpoint, enabling user input.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
    <title>Add User</title>
</head>
<body>
    <h1>Add User</h1>
    <form:form modelAttribute="user" method="post" action="save">
        <table>
            <tr>
                <td>First Name:</td>
```



```

<td><form:input path="firstName"/></td>
</tr>
<tr>
    <td>Last Name:</td>
    <td><form:input path="lastName"/></td>
</tr>
<tr>
    <td>Email:</td>
    <td><form:input path="email"/></td>
</tr>
<tr>
    <td>Active:</td>
    <td><form:checkbox path="active" checked="checked"/></td>
</tr>
<tr>
    <td colspan="2"><input type="submit" value="Save"/></td>
</tr>
</table>
</form:form>
<a href="${pageContext.request.contextPath}/users">Back to User List</a>
</body>
</html>

```

spring-config.xml (located in src/main/resources/): Configures the Spring application, defining the data source for MySQL, entity manager for JPA, transaction manager, and view resolver, ensuring proper integration and runtime behavior.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/mvc
                           http://www.springframework.org/schema/mvc/spring-mvc.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd">

    <!-- Component scanning -->
    <context:component-scan base-package="com.example"/>

    <!-- Enable MVC annotations -->

```



```
<mvc:annotation-driven/>

<!-- Enable transaction management -->
<tx:annotation-driven transaction-manager="transactionManager"/>

<!-- Data Source -->
<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/spring_db"/>
    <property name="username" value="root"/>
    <property name="password" value="Archer@12345"/>
</bean>

<!-- Entity Manager Factory -->
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="packagesToScan" value="com.example.model"/>
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"/>
    </property>
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</prop>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
</bean>

<!-- Transaction Manager -->
<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<!-- JSP View Resolver -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/"/>
    <property name="suffix" value=".jsp"/>
</bean>
</beans>
```



web.xml (located in src/main/webapp/WEB-INF/): Sets up the DispatcherServlet to handle HTTP requests with spring-config.xml as the configuration, serving as the web application descriptor.

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                        http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
    version="4.0">
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

pom.xml (located in project root): Manages project dependencies (Spring, Hibernate, MySQL, Jakarta EE), build plugins, and Java version, facilitating compilation, packaging, and deployment of the application.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.example</groupId>
    <artifactId>spring-jpa-web-crud-jsp-advsq</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>spring-jpa-web-crud-jsp-advsq Maven Webapp</name>
    <url>http://maven.apache.org</url>
    <properties>
        <spring.version>6.2.7</spring.version>
        <java.version>21</java.version>
    </properties>

    <dependencies>
        <!-- Spring Web MVC -->
```



```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-webmvc</artifactId>
<version>${spring.version}</version>
</dependency>
<!-- Spring ORM (for JPA) -->
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-orm</artifactId>
<version>${spring.version}</version>
</dependency>
<!-- Hibernate (JPA Implementation) -->
<dependency>
<groupId>org.hibernate.orm</groupId>
<artifactId>hibernate-core</artifactId>
<version>6.6.1.Final</version>
</dependency>
<!-- MySQL Connector -->
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>8.0.33</version>
</dependency>
<!-- JSP and Servlet API -->
<dependency>
<groupId>jakarta.servlet</groupId>
<artifactId>jakarta.servlet-api</artifactId>
<version>6.0.0</version>
<scope>provided</scope>
</dependency>
<dependency>
<groupId>jakarta.servlet.jsp</groupId>
<artifactId>jakarta.servlet.jsp-api</artifactId>
<version>3.1.0</version>
<scope>provided</scope>
</dependency>
<!-- JSTL -->
<dependency>
<groupId>jakarta.servlet.jsp.jstl</groupId>
<artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
<version>3.0.0</version>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-tx</artifactId>
<version>${spring.version}</version>
</dependency>
```



```
<dependency>
<groupId>org.glassfish.web</groupId>
<artifactId>jakarta.servlet.jsp.jstl</artifactId>
<version>3.0.1</version>
</dependency>
</dependencies>
<build>
<finalName>spring-jpa-web-crud-jsp-advsq</finalName>
</build>
</project>
```

List of URLs

- **/users:**
 - **Purpose:** Displays a list of active users retrieved from the user_summary view. This endpoint leverages the UserSummaryRepository.findByEmail method (with null email to fetch all active users) and renders the result in userList.jsp, ensuring only users with active = 1 are shown.
- **/users/add:**
 - **Purpose:** Serves a form (userForm.jsp) for adding a new user. It initializes a new User object as a model attribute, allowing users to input firstName, lastName, email, and active status, which are then submitted to /users/save.
- **/users/save:**
 - **Purpose:** Handles the submission of the add user form, persisting the User object via UserService.saveUser and the UserRepository.save method. The before_user_insert trigger sets created_at, and it redirects to /users after saving.
- **/users/{id}/fullname:**
 - **Purpose:** Retrieves and displays the full name of a user with the specified id using the get_full_name function. The UserRepository.getFullName method executes a native query, and the result is rendered in userFullName.jsp.
- **/users/{id}/stored:**
 - **Purpose:** Fetches and displays the user's full name and email (formatted as "first_name last_name - email") for the specified id using the get_user_by_id stored procedure. The UserService.callStoredProcedure method invokes the procedure via @NamedStoredProcedureQuery, with the output shown in userFullName.jsp.
- **Database Integration:** Each URL leverages the advanced concepts:
 - /users uses the user_summary view.
 - /users/save triggers the before_user_insert trigger.
 - /users/{id}/fullname uses the get_full_name function.
 - /users/{id}/stored uses the get_user_by_id stored procedure.

Error Analysis:

HTTP Status 500 – Internal Server Error Type Exception Report



Message Request processing failed: java.lang.IllegalArgumentException: Name for argument of type [java.lang.Long] not specified, and parameter name information not available via reflection. Ensure that the compiler uses the '-parameters' flag.

Description The server encountered an unexpected condition that prevented it from fulfilling the request.

Exception

jakarta.servlet.ServletException: Request processing failed:
java.lang.IllegalArgumentException: Name for argument of type [java.lang.Long] not specified, and parameter name information not available via reflection. Ensure that the compiler uses the '-parameters' flag.

In you are using the @PathVariable as

```
@GetMapping("/{id}/fullname")
public String getFullName(@PathVariable Long id, Model model) {
    model.addAttribute("fullName", userService.getFullName(id));
    return "userFullName";
}

@GetMapping("/{id}/stored")
public String callStoredProcedure(@PathVariable Long id, Model model) {
    model.addAttribute("fullName", userService.callStoredProcedure(id));
    return "userFullName";
}
```

Then we need to add plugins in pom.xml as

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.10.1</version>
            <configuration>
                <source>${java.version}</source>
                <target>${java.version}</target>
                <compilerArgs>
                    <arg>-parameters</arg>
                </compilerArgs>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>3.3.2</version>
        </plugin>
    </plugins>
</build>
```



Parameter Names with -parameters:

- The `<compilerArgs><arg>-parameters</arg></compilerArgs>` configuration adds the `-parameters` flag to the Java compiler (`javac`). This flag instructs the compiler to include method parameter names in the bytecode, which is critical for Spring MVC's reflection-based parameter resolution.
- Why It Was Added:** As discussed in the previous error (`IllegalArgumentException: Name for argument of type [java.lang.Long] not specified`), Spring MVC relies on parameter names to bind annotations like `@PathVariable` in methods.

Without the `-parameters` flag, the compiled bytecode does not include the parameter name `id`, causing Spring to fail when resolving the `@PathVariable`. The `-parameters` flag fixes this by embedding parameter names in the bytecode, allowing Spring to infer them via reflection.

- This was one of the solutions (Option 2) provided to fix the `IllegalArgumentException`, alongside explicitly naming the parameter (e.g., `@PathVariable("id")`).

Otherwise alternatively, you have to write explicitly as

```

@GetMapping("/{id}/fullname")
public String getFullName(@PathVariable("id") Long id, Model model) {
    model.addAttribute("fullName", userService.getFullName(id));
    return "userFullName";
}

@GetMapping("/{id}/stored")
public String callStoredProcedure(@PathVariable("id") Long id, Model model) {
    model.addAttribute("fullName", userService.callStoredProcedure(id));
    return "userFullName";
}

```

Create project, integrate it with Spring MVC using Spring6, Java 21, and run on Tomcat 11. Use Thymeleaf For UI. demonstrate `@Entity`, `@Table`, `@Id`, and `@GeneratedValue`, `@Column`, `@Transient`, `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`, `@Transactional`, `@Embeddable` and `@Embedded`, `@PersistenceContext`, use `@NamedQuery` and `@NamedQueries`, `@Inheritance` and `@DiscriminatorColumn`, `@EntityListeners`, `@PrePersist`, `@PostPersist` and `@Version`. use spring-core(not spring boot)

The spring-mvc-jpa project is a web application built using Spring MVC, Spring ORM (JPA with Hibernate), Thymeleaf, and MySQL, deployed on Apache Tomcat 11 with Java 21. It implements a CRUD (Create, Read, Update, Delete) system for managing user data, including associated entities like addresses, profiles, phones, and roles. The application uses a Java-based configuration with `AbstractAnnotationConfigDispatcherServletInitializer` to set up the `DispatcherServlet`, eliminating the need for a `web.xml` file. The frontend is powered by Thymeleaf templates, providing a user-friendly interface for managing users. The project leverages JPA annotations for entity mapping and relationships, Spring annotations for dependency injection and MVC configuration, and transaction management to ensure data integrity. It is designed to run in a Jakarta EE 10/11 environment, ensuring compatibility with modern servlet containers like Tomcat 11.



Key Features:

- **User Management:** Create, view, update, and delete users with details like name, email, address, profile, and phone numbers.
- **JPA Relationships:** Supports @OneToOne, @OneToMany, and @ManyToMany relationships between entities (User, Address, Profile, Phone, Role).
- **Auditing:** Implements auditing with @EntityListeners to track creation and update timestamps.
- **Thymeleaf UI:** Dynamic web pages for listing users, viewing details, and handling form submissions.
- **Java-Based Configuration:** Uses AppConfig.java and DispatcherServletInitializer.java for Spring MVC and JPA setup.
- **Transaction Management:** Ensures data consistency with @Transactional annotations.
- **RESTful URLs:** Intuitive URL mappings for user operations (e.g., /users, /users/{id}).

Application name: spring-jpa-mysql-crud-webapp



Spring-hibernate using Spring-Core

What is Hibernate?

Hibernate is an open-source Object-Relational Mapping (ORM) framework for Java that simplifies database interactions by mapping Java objects to database tables. It provides a way to perform CRUD (Create, Read, Update, Delete) operations on database entities using Java objects, abstracting away much of the low-level JDBC (Java Database Connectivity) and SQL code. Hibernate handles the conversion between Java objects and database records, manages database connections, and supports features like caching, lazy loading, and transaction management.

Key features of Hibernate include:

- **Object-Relational Mapping:** Maps Java classes to database tables and Java fields to table columns.
- **HQL (Hibernate Query Language):** A powerful, object-oriented query language similar to SQL but operates on Java objects.
- **Automatic Table Management:** Generates and manages database schema based on Java entity classes.
- **Caching:** Supports first-level and second-level caching to improve performance.
- **Transaction Management:** Integrates with Java Transaction API (JTA) and supports ACID transactions.
- **Lazy and Eager Loading:** Controls how related data is fetched from the database.

Difference Between JPA and Hibernate

JPA (Java Persistence API) and **Hibernate** are closely related but serve different purposes. Here's a clear comparison:

Aspect	JPA	Hibernate
Definition	JPA is a specification that defines a standard for ORM in Java applications.	Hibernate is an implementation of the JPA specification and a standalone ORM framework.
Nature	A set of interfaces and annotations provided by Java EE (Jakarta EE).	A concrete framework that implements JPA and adds its own features.
Scope	Provides a standard API for ORM, but does not include implementation logic.	Extends JPA with additional features like advanced caching, HQL, and more.
Portability	Highly portable; applications using JPA can switch between providers (e.g., Hibernate, EclipseLink).	Less portable; Hibernate-specific features tie the application to Hibernate.
Features	Defines core ORM functionalities like entity management, JPQL (Java Persistence Query Language), and transactions.	Includes JPA features plus extras like HQL, second-level caching, and custom mappings.
Query Language	Uses JPQL, a standardized query language similar to SQL.	Uses HQL (extends JPQL) and supports native SQL queries.
Configuration	Configured via persistence.xml and annotations like @Entity, @Id.	Extends JPA configuration with Hibernate-specific properties and annotations.
Extensibility	Limited to the standard API; no proprietary features.	Offers advanced features like Criteria API, filters, and interceptors.



Dependency	Included in Java EE or added via libraries like javax.persistence.	Requires Hibernate libraries (e.g., hibernate-core) in addition to JPA.
Examples	Used with any JPA-compliant provider (Hibernate, EclipseLink, OpenJPA).	Used as a JPA provider or standalone with Hibernate-specific features.

Key Points to Understand

1. **JPA is a Specification, Hibernate is an Implementation:**
 - JPA defines a standard set of interfaces and annotations for ORM, but it doesn't provide the actual code to perform database operations.
 - Hibernate implements JPA, meaning it adheres to JPA standards but also adds its own proprietary features.
2. **When to Use JPA vs. Hibernate:**
 - Use **JPA** if you want portability across different ORM providers (e.g., switching from Hibernate to EclipseLink without changing code).
 - Use **Hibernate** if you need advanced features like second-level caching, complex mappings, or HQL, but this may lock you into Hibernate.
3. **Example:**
 - **JPA Code (Standard):**

```
@Entity
public class User {
    @Id
    private Long id;
    private String name;
    // Getters and setters
}
```

```
EntityManager em = entityManagerFactory.createEntityManager();
em.persist(new User(1L, "Alice"));
```

- **Hibernate-Specific Code:**

```
Session session = sessionFactory.openSession();
session.save(new User(1L, "Alice"));
Query query = session.createQuery("from User where name = :name", User.class);
query.setParameter("name", "Alice");
```

4. **Relationship:**
 - Hibernate is a superset of JPA. You can use Hibernate as a JPA provider (adhering to JPA standards) or use its native APIs for additional functionality.
 - If you stick to JPA standards, your code remains portable. Using Hibernate-specific features (e.g., HQL or Session API) makes your code dependent on Hibernate.

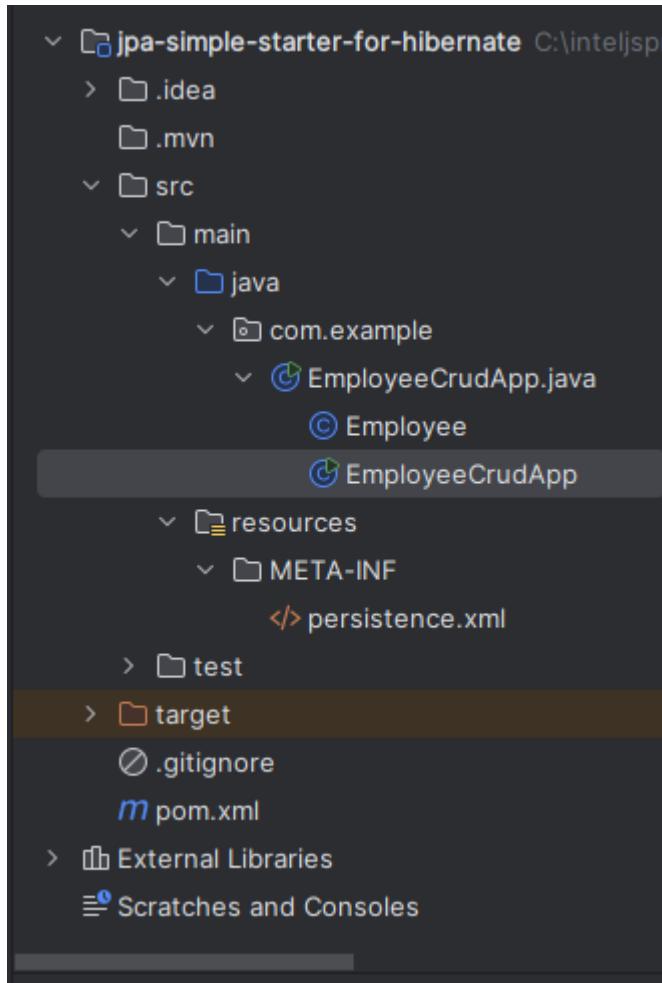
Let's start With Simple JPA CRUD (Without Spring Framework)

Below is a single-file Java program that implements a menu-driven console-based CRUD (Create, Read, Update, Delete) application using JPA (Java Persistence API) with MySQL as the database. The program manages a simple Employee entity and uses Hibernate as the JPA provider. The code



includes a persistence.xml configuration file (as an artifact) and assumes a MySQL database is set up locally.

```
CREATE DATABASE hb_employee_db;
```



Pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>jpa-simple-starter-for-hibernate</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>jpa-simple-starter-for-hibernate</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
```



```
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.6.15.Final</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.33</version>
    </dependency>
    <dependency>
        <groupId>javax.persistence</groupId>
        <artifactId>javax.persistence-api</artifactId>
        <version>2.2</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
</project>
```

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
    <persistence-unit name="EmployeePU" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <class>com.example.Employee</class>
        <properties>
            <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/hb_employee_db"/>
            <property name="javax.persistence.jdbc.user" value="root"/>
            <property name="javax.persistence.jdbc.password" value="Archer@12345"/>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```



EmployeeCrudApp.java

```
package com.example;
import javax.persistence.*;
import java.util.List;
import java.util.Scanner;

@Entity
@Table(name = "employees")
class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private double salary;

    // Default constructor
    public Employee() {}

    // Parameterized constructor
    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    // Getters and setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public double getSalary() { return salary; }
    public void setSalary(double salary) { this.salary = salary; }

    @Override
    public String toString() {
        return "Employee{id=" + id + ", name='" + name + "', salary=" + salary + "}";
    }
}

public class EmployeeCrudApp {
    private static EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("EmployeePU");

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("\n== Employee Management System ==");
            System.out.println("1. Create Employee");
            System.out.println("2. Read All Employees");
            System.out.println("3. Update Employee");
            System.out.println("4. Delete Employee");
            System.out.println("5. Exit");
        }
    }
}
```



```
System.out.print("Choose an option: ");
int choice = scanner.nextInt();
scanner.nextLine(); // Consume newline

switch (choice) {
    case 1:
        createEmployee(scanner);
        break;
    case 2:
        readAllEmployees();
        break;
    case 3:
        updateEmployee(scanner);
        break;
    case 4:
        deleteEmployee(scanner);
        break;
    case 5:
        emf.close();
        scanner.close();
        System.out.println("Exiting...");
        return;
    default:
        System.out.println("Invalid option. Try again.");
}
}

private static void createEmployee(Scanner scanner) {
    System.out.print("Enter employee name: ");
    String name = scanner.nextLine();
    System.out.print("Enter employee salary: ");
    double salary = scanner.nextDouble();

    EntityManager em = emf.createEntityManager();
    try {
        em.getTransaction().begin();
        Employee employee = new Employee(name, salary);
        em.persist(employee);
        em.getTransaction().commit();
        System.out.println("Employee created successfully!");
    } catch (Exception e) {
        em.getTransaction().rollback();
        System.out.println("Error creating employee: " + e.getMessage());
    } finally {
        em.close();
    }
}

private static void readAllEmployees() {
    EntityManager em = emf.createEntityManager();
```



```
try {
    TypedQuery<Employee> query = em.createQuery("SELECT e FROM Employee e",
Employee.class);
    List<Employee> employees = query.getResultList();
    if (employees.isEmpty()) {
        System.out.println("No employees found.");
    } else {
        System.out.println("Employee List:");
        for (Employee emp : employees) {
            System.out.println(emp);
        }
    }
} catch (Exception e) {
    System.out.println("Error reading employees: " + e.getMessage());
} finally {
    em.close();
}
```

```
private static void updateEmployee(Scanner scanner) {
    System.out.print("Enter employee ID to update: ");
    Long id = scanner.nextLong();
    scanner.nextLine(); // Consume newline

    EntityManager em = emf.createEntityManager();
    try {
        em.getTransaction().begin();
        Employee employee = em.find(Employee.class, id);
        if (employee == null) {
            System.out.println("Employee not found.");
        } else {
            System.out.print("Enter new name (leave blank to keep '" + employee.getName() + "'): ");
            String name = scanner.nextLine();
            if (!name.isEmpty()) {
                employee.setName(name);
            }
            System.out.print("Enter new salary (enter 0 to keep " + employee.getSalary() + "): ");
            double salary = scanner.nextDouble();
            if (salary > 0) {
                employee.setSalary(salary);
            }
            em.getTransaction().commit();
            System.out.println("Employee updated successfully!");
        }
    } catch (Exception e) {
        em.getTransaction().rollback();
        System.out.println("Error updating employee: " + e.getMessage());
    } finally {
        em.close();
    }
}
```



```

private static void deleteEmployee(Scanner scanner) {
    System.out.print("Enter employee ID to delete: ");
    Long id = scanner.nextLong();

    EntityManager em = emf.createEntityManager();
    try {
        em.getTransaction().begin();
        Employee employee = em.find(Employee.class, id);
        if (employee == null) {
            System.out.println("Employee not found.");
        } else {
            em.remove(employee);
            em.getTransaction().commit();
            System.out.println("Employee deleted successfully!");
        }
    } catch (Exception e) {
        em.getTransaction().rollback();
        System.out.println("Error deleting employee: " + e.getMessage());
    } finally {
        em.close();
    }
}
}

```

Explanation (Not Part of Response)

- **Employee Entity:** A simple JPA entity with id, name, and salary fields, annotated with @Entity, @Id, and @GeneratedValue.
- **Menu-Driven Interface:** The program runs in a loop, prompting the user to choose from five options: create, read, update, delete, or exit.
- **CRUD Operations:**
 - **Create:** Prompts for name and salary, persists a new Employee object.
 - **Read:** Retrieves and displays all employees using a JPQL query.
 - **Update:** Finds an employee by ID, allows updating name and/or salary.
 - **Delete:** Removes an employee by ID.
- **Persistence Unit:** Configured in persistence.xml to connect to MySQL, using Hibernate as the JPA provider. The hbm2ddl.auto=update property automatically creates/updates the database schema.
- **Error Handling:** Each operation is wrapped in a try-catch block to handle exceptions and rollback transactions if needed.
- **Single File:** The Employee entity and the main application logic are in one file (EmployeeCrudApp.java) as requested.

Let's Convert above example entirely in hibernate style

Below is a single-file Java program that implements a menu-driven console-based CRUD (Create, Read, Update, Delete) application using Hibernate's native API instead of JPA. The program manages an Employee entity and connects to a MySQL database. It includes a Hibernate configuration file (hibernate.cfg.xml) as an artifact and assumes the same MySQL setup as the previous example.



Assumptions:

- MySQL database is running locally with a schema named hb_employee_db.
- MySQL username: root, password: password (update as per your setup).
- Required dependencies (Hibernate and MySQL Connector) are included.
- The program uses a single Employee entity with fields: id, name, and salary.

Instructions:

1. Set up a MySQL database:

```
CREATE DATABASE employee_db;
```

2. Add dependencies to your pom.xml (for Maven) or equivalent for Gradle:

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.6.15.Final</version>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
  </dependency>
</dependencies>
```

3. Place the hibernate.cfg.xml file (provided below) in src/main/resources/.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hb_employee_db</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">Archer@12345</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
    <property name="hibernate.hbm2ddl.auto">update</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.format_sql">true</property>
    <mapping class="com.example.Employee"/>
  </session-factory>
</hibernate-configuration>
```

4. Pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
```



```
<artifactId>hibernate-simple-starter-for-spring-hibernate</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>

<name>hibernate-simple-starter-for-spring-hibernate</name>
<url>http://maven.apache.org</url>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.6.15.Final</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.33</version>
    </dependency>
</dependencies>
</project>
```

5. Main class

```
package com.example;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.query.Query;

import javax.persistence.*;
import java.util.List;
import java.util.Scanner;

@Entity
@Table(name = "employees")
class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private double salary;

    // Default constructor
    public Employee() {}

    // Parameterized constructor
    public Employee(String name, double salary) {
```



```
this.name = name;
this.salary = salary;
}

// Getters and setters
public Long getId() { return id; }
public void setId(Long id) { this.id = id; }
public String getName() { return name; }
public void setName(String name) { this.name = name; }
public double getSalary() { return salary; }
public void setSalary(double salary) { this.salary = salary; }

@Override
public String toString() {
    return "Employee{id=" + id + ", name='" + name + "', salary=" + salary + "}";
}
}

public class EmployeeHibernateApp {
    private static SessionFactory sessionFactory;

    static {
        try {
            sessionFactory = new Configuration()
                .configure("hibernate.cfg.xml")
                .addAnnotatedClass(Employee.class)
                .buildSessionFactory();
        } catch (Exception e) {
            System.err.println("Error initializing Hibernate: " + e.getMessage());
            throw new ExceptionInInitializerError(e);
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("\n==== Employee Management System (Hibernate) ====");
            System.out.println("1. Create Employee");
            System.out.println("2. Read All Employees");
            System.out.println("3. Update Employee");
            System.out.println("4. Delete Employee");
            System.out.println("5. Exit");
            System.out.print("Choose an option: ");
            int choice = scanner.nextInt();
            scanner.nextLine(); // Consume newline

            switch (choice) {
                case 1:
                    createEmployee(scanner);
                    break;
                case 2:
```



```
        readAllEmployees();
        break;
    case 3:
        updateEmployee(scanner);
        break;
    case 4:
        deleteEmployee(scanner);
        break;
    case 5:
        sessionFactory.close();
        scanner.close();
        System.out.println("Exiting...");
        return;
    default:
        System.out.println("Invalid option. Try again.");
    }
}
}

private static void createEmployee(Scanner scanner) {
    System.out.print("Enter employee name: ");
    String name = scanner.nextLine();
    System.out.print("Enter employee salary: ");
    double salary = scanner.nextDouble();

    Session session = sessionFactory.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        Employee employee = new Employee(name, salary);
        session.save(employee);
        tx.commit();
        System.out.println("Employee created successfully!");
    } catch (Exception e) {
        if (tx != null) tx.rollback();
        System.out.println("Error creating employee: " + e.getMessage());
    } finally {
        session.close();
    }
}

private static void readAllEmployees() {
    Session session = sessionFactory.openSession();
    try {
        Query<Employee> query = session.createQuery("FROM Employee", Employee.class);
        List<Employee> employees = query.getResultList();
        if (employees.isEmpty()) {
            System.out.println("No employees found.");
        } else {
            System.out.println("Employee List:");
            for (Employee emp : employees) {
```



```
        System.out.println(emp);
    }
}
} catch (Exception e) {
    System.out.println("Error reading employees: " + e.getMessage());
} finally {
    session.close();
}
}

private static void updateEmployee(Scanner scanner) {
    System.out.print("Enter employee ID to update: ");
    Long id = scanner.nextLong();
    scanner.nextLine(); // Consume newline

    Session session = sessionFactory.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        Employee employee = session.get(Employee.class, id);
        if (employee == null) {
            System.out.println("Employee not found.");
        } else {
            System.out.print("Enter new name (leave blank to keep '" + employee.getName() + "'): ");
            String name = scanner.nextLine();
            if (!name.isEmpty()) {
                employee.setName(name);
            }
            System.out.print("Enter new salary (enter 0 to keep " + employee.getSalary() + "): ");
            double salary = scanner.nextDouble();
            if (salary > 0) {
                employee.setSalary(salary);
            }
            session.update(employee);
            tx.commit();
            System.out.println("Employee updated successfully!");
        }
    } catch (Exception e) {
        if (tx != null) tx.rollback();
        System.out.println("Error updating employee: " + e.getMessage());
    } finally {
        session.close();
    }
}

private static void deleteEmployee(Scanner scanner) {
    System.out.print("Enter employee ID to delete: ");
    Long id = scanner.nextLong();

    Session session = sessionFactory.openSession();
    Transaction tx = null;
```



```
try {
    tx = session.beginTransaction();
    Employee employee = session.get(Employee.class, id);
    if (employee == null) {
        System.out.println("Employee not found.");
    } else {
        session.delete(employee);
        tx.commit();
        System.out.println("Employee deleted successfully!");
    }
} catch (Exception e) {
    if (tx != null) tx.rollback();
    System.out.println("Error deleting employee: " + e.getMessage());
} finally {
    session.close();
}
}
```

Changes from JPA to Hibernate Style

- 1. SessionFactory Instead of EntityManagerFactory:**
 - Replaced EntityManagerFactory with Hibernate's SessionFactory, configured using hibernate.cfg.xml.
 - Used Configuration().configure() to load Hibernate settings and register the Employee class.
 - 2. Session Instead of EntityManager:**
 - Replaced EntityManager with Session for database operations.
 - Used session.save(), session.update(), session.delete(), and session.get() instead of JPA's persist(), merge(), remove(), and find().
 - 3. HQL Instead of JPQL:**
 - Used Hibernate's HQL (FROM Employee) in the readAllEmployees method, though the syntax is identical to JPQL in this case.
 - 4. Transaction Management:**
 - Used Hibernate's Transaction (session.beginTransaction()) instead of JPA's EntityTransaction.
 - 5. Configuration:**
 - Replaced persistence.xml with hibernate.cfg.xml, which is specific to Hibernate's native API.
 - Maintained the same database connection settings and hbm2ddl.auto=update for automatic schema management.

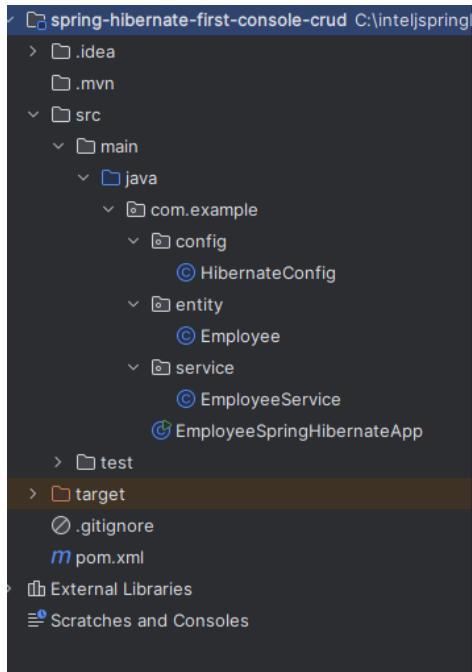
Same Application in spring Framework using Spring-Hibernate:

The **Employee Management System (Spring-Hibernate)** is a console-based Java application designed to perform CRUD (Create, Read, Update, Delete) operations on employee records stored in a MySQL database. It leverages **Spring Core** for dependency injection and **Spring-Hibernate** integration for ORM (Object-Relational Mapping) and transaction management. The application features a menu-driven interface, allowing users to manage employee data interactively. The project is modular, with separate classes for the entity, service layer, configuration, and main application, ensuring maintainability and adherence to Spring best practices.



Project Folder Structure

Below is the project folder structure based on the provided files, assuming a Maven-based project:



Introduction to the Project

The **Employee Management System (Spring-Hibernate)** is a robust console application built to demonstrate the integration of Spring and Hibernate for managing employee data in a MySQL database. The application provides a user-friendly, menu-driven interface to perform essential CRUD operations: creating new employee records, retrieving all employees, updating existing records, and deleting employees. By utilizing Spring's dependency injection and transaction management, combined with Hibernate's ORM capabilities, the project ensures efficient database interactions and maintainable code. The modular design separates concerns into entity, service, configuration, and application layers, making it scalable and easy to extend.

Key features include:

- **Spring Core:** Manages beans and dependencies via dependency injection.
- **Spring-Hibernate Integration:** Simplifies Hibernate configuration and transaction management using Spring's LocalSessionFactoryBean and @Transactional annotations.
- **Hibernate ORM:** Maps the Employee entity to a database table, abstracting SQL operations.
- **Menu-Driven Interface:** Allows users to interact with the application through a console-based menu.
- **MySQL Database:** Stores employee data in the hb_employee_db database.

This project is ideal for learning Spring-Hibernate integration, understanding dependency injection, and implementing transactional database operations in a Java application.

File List with Roles

Below is the list of provided files, their locations, and their roles in the project:



1. EmployeeSpringHibernateApp.java

- **Location:** src/main/java/com/example/EmployeeSpringHibernateApp.java
- **Role:** The main application class containing the entry point (main method). It initializes the Spring application context using AnnotationConfigApplicationContext and the HibernateConfig class. It provides a menu-driven console interface for users to interact with the application, delegating CRUD operations to the EmployeeService bean. The class orchestrates user input handling and application lifecycle management (e.g., closing the context).

```
package com.example;

import com.example.config.HibernateConfig;
import com.example.service.EmployeeService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import java.util.Scanner;

public class EmployeeSpringHibernateApp {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(HibernateConfig.class);
        EmployeeService employeeService = context.getBean(EmployeeService.class);
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("\n==== Employee Management System (Spring-Hibernate) ===");
            System.out.println("1. Create Employee");
            System.out.println("2. Read All Employees");
            System.out.println("3. Update Employee");
            System.out.println("4. Delete Employee");
            System.out.println("5. Exit");
            System.out.print("Choose an option: ");
            int choice = scanner.nextInt();
            scanner.nextLine(); // Consume newline

            switch (choice) {
                case 1:
                    System.out.print("Enter employee name: ");
                    String name = scanner.nextLine();
                    System.out.print("Enter employee salary: ");
                    double salary = scanner.nextDouble();
                    employeeService.createEmployee(name, salary);
                    break;
                case 2:
                    employeeService.readAllEmployees();
            }
        }
    }
}
```



```
        break;
    case 3:
        System.out.print("Enter employee ID to update: ");
        Long id = scanner.nextLong();
        scanner.nextLine(); // Consume newline
        System.out.print("Enter new name (leave blank to keep unchanged): ");
        String newName = scanner.nextLine();
        System.out.print("Enter new salary (enter 0 to keep unchanged): ");
        double newSalary = scanner.nextDouble();
        employeeService.updateEmployee(id, newName, newSalary);
        break;
    case 4:
        System.out.print("Enter employee ID to delete: ");
        id = scanner.nextLong();
        employeeService.deleteEmployee(id);
        break;
    case 5:
        context.close();
        scanner.close();
        System.out.println("Exiting...");
        return;
    default:
        System.out.println("Invalid option. Try again.");
    }
}
}
```

2. Employee.java

- **Location:** src/main/java/com/example/entity/Employee.java
 - **Role:** The entity class representing an employee in the database. Annotated with JPA annotations (@Entity, @Table, @Id, @GeneratedValue), it maps to the employees table in the hb_employee_db database. It defines fields (id, name, salary), constructors, getters, setters, and a `toString` method for data representation.

```
package com.example.entity;
import javax.persistence.*;
import java.util.Date;

@Entity
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private double salary;
}
```



```

public Employee() {}

public Employee(String name, double salary) {
    this.name = name;
    this.salary = salary;
}

public Long getId() { return id; }
public void setId(Long id) { this.id = id; }
public String getName() { return name; }
public void setName(String name) { this.name = name; }
public double getSalary() { return salary; }
public void setSalary(double salary) { this.salary = salary; }

@Override
public String toString() {
    return "Employee{id=" + id + ", name='" + name + "', salary=" + salary + "}";
}
}

```

3. HibernateConfig.java

- **Location:** src/main/java/com/example/config/HibernateConfig.java
- **Role:** The Spring configuration class annotated with @Configuration and @EnableTransactionManagement. It defines beans for:
 - DataSource: Configures the MySQL database connection.
 - LocalSessionFactoryBean: Sets up the Hibernate SessionFactory, scanning the com.example package for entities.
 - HibernateTransactionManager: Manages transactions for Hibernate operations.
 - EmployeeService: Instantiates the service layer with the SessionFactory. It also configures Hibernate properties (e.g., dialect, schema update, SQL logging).

```

package com.example.config;

import com.example.service.EmployeeService;
import org.hibernate.SessionFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.hibernate5.HibernateTransactionManager;
import org.springframework.orm.hibernate5.LocalSessionFactoryBean;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;
import java.util.Properties;

@Configuration

```



```
@EnableTransactionManagement
public class HibernateConfig {
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/hb_employee_db");
        dataSource.setUsername("root");
        dataSource.setPassword("Archer@12345");
        return dataSource;
    }

    @Bean
    public LocalSessionFactoryBean sessionFactory(DataSource dataSource) {
        LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
        sessionFactory.setDataSource(dataSource);
        sessionFactory.setPackagesToScan("com.example");
        sessionFactory.setHibernateProperties(hibernateProperties());
        return sessionFactory;
    }

    private Properties hibernateProperties() {
        Properties props = new Properties();
        props.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQL8Dialect");
        props.setProperty("hibernate.hbm2ddl.auto", "update");
        props.setProperty("hibernate.show_sql", "true");
        props.setProperty("hibernate.format_sql", "true");
        return props;
    }

    @Bean
    public HibernateTransactionManager transactionManager(SessionFactory sessionFactory) {
        HibernateTransactionManager txManager = new HibernateTransactionManager();
        txManager.setSessionFactory(sessionFactory);
        return txManager;
    }

    @Bean
    public EmployeeService employeeService(SessionFactory sessionFactory) {
        return new EmployeeService(sessionFactory);
    }
}
```

4. EmployeeService.java

- **Location:** src/main/java/com/example/service/EmployeeService.java
- **Role:** The service layer class responsible for encapsulating business logic and database operations. It uses the Hibernate SessionFactory to perform CRUD



operations on the Employee entity. Methods are annotated with @Transactional to manage database transactions declaratively. The class provides methods for creating, reading, updating, and deleting employees, with console output for user feedback.

```
package com.example.service;
import com.example.entity.Employee;
import org.hibernate.SessionFactory;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

public class EmployeeService {
    private final SessionFactory sessionFactory;

    public EmployeeService(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    @Transactional
    public void createEmployee(String name, double salary) {
        Employee employee = new Employee(name, salary);
        sessionFactory.getCurrentSession().save(employee);
        System.out.println("Employee created successfully!");
    }

    @Transactional(readOnly = true)
    public void readAllEmployees() {
        List<Employee> employees = sessionFactory.getCurrentSession()
            .createQuery("FROM Employee", Employee.class)
            .getResultList();
        if (employees.isEmpty()) {
            System.out.println("No employees found.");
        } else {
            System.out.println("Employee List:");
            for (Employee emp : employees) {
                System.out.println(emp);
            }
        }
    }

    @Transactional
    public void updateEmployee(Long id, String name, double salary) {
        Employee employee = sessionFactory.getCurrentSession().get(Employee.class, id);
        if (employee == null) {
            System.out.println("Employee not found.");
        } else {
            if (!name.isEmpty()) {
                employee.setName(name);
            }
        }
    }
}
```



```
        }
        if (salary > 0) {
            employee.setSalary(salary);
        }
        sessionFactory.getCurrentSession().update(employee);
        System.out.println("Employee updated successfully!");
    }
}

@Transactional
public void deleteEmployee(Long id) {
    Employee employee = sessionFactory.getCurrentSession().get(Employee.class, id);
    if (employee == null) {
        System.out.println("Employee not found.");
    } else {
        sessionFactory.getCurrentSession().delete(employee);
        System.out.println("Employee deleted successfully!");
    }
}
}
```

5. pom.xml (Not Provided, but Assumed)

- **Location:** employee-spring-hibernate-crud/pom.xml
- **Role:** The Maven configuration file defining project dependencies (Spring Core, Spring ORM, Hibernate, MySQL Connector), build plugins, and project metadata. It ensures the necessary libraries are included for compilation and runtime.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>spring-hibernate-first-console-crud</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>spring-hibernate-first-console-crud</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <!-- Spring Core -->
```



```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>5.3.34</version>
</dependency>
<!-- Spring ORM for Hibernate integration -->
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-orm</artifactId>
<version>5.3.34</version>
</dependency>
<!-- Hibernate Core -->
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-core</artifactId>
<version>5.6.15.Final</version>
</dependency>
<!-- MySQL Connector -->
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>8.0.33</version>
</dependency>
</dependencies>
</project>
```

Notes

- **Package Structure:** The files use a modular package structure:
 - com.example.entity: For the Employee entity.
 - com.example.service: For the EmployeeService class.
 - com.example.config: For the HibernateConfig class.
 - com.example: For the main application (EmployeeSpringHibernateApp).
- **Database Setup:** The application assumes a MySQL database hb_employee_db with connection details (URL: jdbc:mysql://localhost:3306/hb_employee_db, username: root, password: Archer@12345). Create the database with:
CREATE DATABASE hb_employee_db;
- **Resources Directory:** No resource files (e.g., application.properties or XML configs) were provided, but src/main/resources/ is typically used for such files. The Java-based configuration in HibernateConfig.java eliminates the need for a separate hibernate.cfg.xml.

Difference between spring-jpa and spring-hibernate (both using spring-core)

The terms **Spring-JPA** and **Spring-Hibernate** (both using Spring Core) often cause confusion because they overlap significantly, as both involve using Spring Core to integrate with a persistence layer. However, there are key differences in their focus, configuration, and usage. Below, I'll clarify the differences concisely, leveraging your background in Basic JPA, Hibernate, Spring-JDBC, and Spring-JPA.



1. Conceptual Overview

- **Spring-JPA:**
 - Refers to Spring's integration with the **Java Persistence API (JPA)**, a standard specification for ORM (Object-Relational Mapping) in Java.
 - Spring-JPA uses Spring Core to provide abstractions (e.g., EntityManager, repositories) for working with any JPA-compliant provider (e.g., Hibernate, EclipseLink).
 - It emphasizes **Spring Data JPA**, which simplifies data access with repository interfaces, reducing boilerplate code.
 - Focuses on a standardized, provider-agnostic approach to persistence.
- **Spring-Hibernate:**
 - Refers to Spring's direct integration with **Hibernate**, a specific ORM framework that implements the JPA specification but also offers native Hibernate features.
 - Uses Spring Core to configure Hibernate's SessionFactory and manage Hibernate-specific sessions and transactions.
 - Allows access to Hibernate's proprietary features (e.g., HQL, Criteria API, second-level caching) that go beyond the JPA standard.
 - More tightly coupled to Hibernate as the ORM provider.

2. Key Differences

Aspect	Spring-JPA	Spring-Hibernate
Underlying Technology	Uses JPA specification, with Hibernate or another provider as the implementation.	Directly uses Hibernate's native APIs (Session, SessionFactory).
Configuration	Configured with EntityManagerFactory (e.g., LocalContainerEntityManagerFactoryBean).	Configured with SessionFactory (e.g., LocalSessionFactoryBean).
Data Access	Primarily uses Spring Data JPA repositories (@Repository, CrudRepository).	Uses Hibernate's Session or native APIs (HQL, Criteria) for data access.
Flexibility	Provider-agnostic; can switch between Hibernate, EclipseLink, etc.	Tied to Hibernate, leveraging its proprietary features.
Features	Limited to JPA standard features (e.g., JPQL, entity mappings).	Access to Hibernate-specific features like HQL, Criteria API, and caching.
Transaction Management	Uses JPA's EntityManager with @Transactional for transaction management.	Uses Hibernate's Session with @Transactional for transaction management.
Learning Curve	Simpler due to Spring Data JPA's abstractions and less boilerplate.	Steeper due to Hibernate's native APIs and configuration complexity.
Use Case	Preferred for standardized, portable applications with minimal provider-specific code.	Preferred when Hibernate's advanced features (e.g., caching, batching) are needed.

3. Practical Differences (Using Spring Core)

- **Spring-JPA (with Spring Core):**



- Configuration typically involves setting up a LocalContainerEntityManagerFactoryBean in Spring Core to create an EntityManagerFactory.
- Example configuration:

```
@Bean
```

```
public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource) {  
    LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();  
    em.setDataSource(dataSource);  
    em.setPackagesToScan("com.example.entity");  
    em.setJpaVendorAdapter(new HibernateJpaVendorAdapter());  
    return em;  
}
```

- Data access is simplified with Spring Data JPA repositories:

```
@Repository
```

```
public interface UserRepository extends JpaRepository<User, Long> {  
    List<User> findByName(String name);  
}
```

- Transactions are managed via JPA's EntityManager with Spring's @Transactional.
- Ideal when you want to write portable code that isn't tied to a specific ORM provider.

- **Spring-Hibernate (with Spring Core):**

- Configuration involves setting up a LocalSessionFactoryBean to create a Hibernate SessionFactory.
- Example configuration:

```
@Bean
```

```
public LocalSessionFactoryBean sessionFactory(DataSource dataSource) {  
    LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();  
    sessionFactory.setDataSource(dataSource);  
    sessionFactory.setPackagesToScan("com.example.entity");  
    sessionFactory.setHibernateProperties(hibernateProperties());  
    return sessionFactory;  
}
```

- Data access uses Hibernate's native Session or APIs like HQL/Criteria:

```
@Autowired
```

```
private SessionFactory sessionFactory;
```

```
public List<User> findByName(String name) {
```

```
    Session session = sessionFactory.getCurrentSession();  
    Query<User> query = session.createQuery("from User where name = :name", User.class);  
    query.setParameter("name", name);  
    return query.getResultList();
```

```
}
```

- Transactions are managed via Hibernate's Session with Spring's @Transactional.



- Suited for applications requiring Hibernate-specific features like second-level caching or complex HQL queries.
-

4. When to Use Which

- **Use Spring-JPA:**
 - When you want a standardized, portable solution that works with any JPA provider.
 - When leveraging Spring Data JPA's repository abstractions to reduce boilerplate.
 - For simpler applications where JPA's standard features are sufficient.
 - Example: Rapid development of CRUD applications with minimal configuration.
 - **Use Spring-Hibernate:**
 - When you need Hibernate's advanced features (e.g., second-level caching, batch processing, or Criteria API).
 - For complex applications requiring fine-grained control over persistence logic.
 - When optimizing performance with Hibernate-specific techniques (e.g., avoiding N+1 issues).
 - Example: Enterprise applications with complex data models and high-performance requirements.
-

5. Hybrid Approach

- In practice, Spring-JPA often uses Hibernate as the underlying JPA provider (configured via `HibernateJpaVendorAdapter`). This allows you to combine Spring Data JPA's simplicity with Hibernate's advanced features when needed.
 - Example: Use Spring Data JPA repositories for most CRUD operations but fall back to Hibernate's Session for complex queries or caching.
-

Summary

- **Spring-JPA** is a higher-level abstraction that uses JPA (often with Hibernate underneath) and Spring Data JPA for simplicity and portability.
 - **Spring-Hibernate** directly uses Hibernate's native APIs, offering more control and access to Hibernate-specific features but with increased complexity.
 - Both leverage Spring Core for dependency injection and transaction management, but Spring-JPA is more abstracted, while Spring-Hibernate is more hands-on with Hibernate's capabilities.
-

Rewrite the above application, write it as web application and use JSP for UI

Rewrite the above application, write it as web application and use Thymeleaf for UI

