Spring JDBC

1. Introduction to Spring JDBC

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

Introduction to Spring JDBC

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

1. Limitations of Traditional JDBC

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

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

2. Advantages of Spring JDBC

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

- **Reduced Boilerplate Code**: Spring JDBC eliminates repetitive code by providing utilities like JdbcTemplate, which handles resource management and simplifies query execution.
- **Simplified Exception Handling**: Spring JDBC wraps SQLException into unchecked DataAccessException hierarchies, making exception handling cleaner and more consistent across databases.
- **Resource Management**: Spring automatically manages database connections, statements, and result sets, ensuring resources are properly closed to prevent leaks.
- **Database Portability**: Spring abstracts vendor-specific error codes into a consistent exception hierarchy, improving portability across different databases.
- **Simplified Transaction Management**: Spring provides declarative and programmatic transaction management, reducing the need for manual transaction handling.

- **Support for Common Operations**: Spring JDBC supports common database operations like querying, updating, batch processing, and mapping results to Java objects with minimal code.
- **Integration with Spring Ecosystem**: Spring JDBC integrates seamlessly with other Spring modules (e.g., Spring ORM, Spring Transaction) and dependency injection, enabling modular and testable applications.

---

3. Key Features of Spring JDBC

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

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

---

4. Overview of JdbcTemplate and Spring DataSource Support

JdbcTemplate

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

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

```
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

public class JdbcTemplateExample {
    public static void main(String[] args) {
```

```
        // Configure DataSource
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/mydb");
        dataSource.setUsername("root");
        dataSource.setPassword("password");

        // Initialize JdbcTemplate
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        // Execute a query
        String sql = "SELECT name FROM employee WHERE id = ?";
        String name = jdbcTemplate.queryForObject(sql, String.class, 1);
        System.out.println("Employee Name: " + name);
    }
}
```

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

Spring DataSource Support

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

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

- **Configuration Example (Using Spring XML)**:

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
    <property name="username" value="root"/>
    <property name="password" value="password"/>
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

- **Benefits of DataSource Support**:
  - Centralized configuration of database connection details.
  - Integration with connection pools for efficient resource usage.

- Support for dependency injection, allowing easy configuration in Spring applications.
- Simplifies switching between different database environments (e.g., development, testing, production).

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

---

2. Setting Up Spring JDBC Environment

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

Setting Up Spring JDBC Environment

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

---

1. Required Dependencies (Maven/Gradle)

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

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

```
<dependencies>
  <!-- Spring JDBC -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>6.1.13</version> <!-- Use the latest version -->
  </dependency>

  <!-- JDBC Driver (e.g., MySQL) -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version> <!-- Use the latest version -->
```


```
    </dependency>

    <!-- Optional: HikariCP for connection pooling -->
    <dependency>
        <groupId>com.zaxxer</groupId>
        <artifactId>HikariCP</artifactId>
        <version>5.1.0</version> <!-- Use the latest version -->
    </dependency>

    <!-- Optional: Apache DBCP2 for connection pooling -->
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-dbcp2</artifactId>
        <version>2.12.0</version> <!-- Use the latest version -->
    </dependency>
</dependencies>
```

Notes:

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

---

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

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

XML-Based Configuration

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

Example: XML Configuration

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

    <!-- DataSource Configuration -->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
        <property name="username" value="root"/>
        <property name="password" value="password"/>
    </bean>
```


```
    <!-- JdbcTemplate Configuration -->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
    </bean>
</beans>
```

Usage:

- Load the XML configuration in your application:

```
ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
JdbcTemplate jdbcTemplate = context.getBean("jdbcTemplate", JdbcTemplate.class);
```

Notes:

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

Java-Based Configuration (AnnotationConfig)

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

Example: Java Configuration

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

import javax.sql.DataSource;
@Configuration
public class SpringConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/mydb");
        dataSource.setUsername("root");
        dataSource.setPassword("password");
        return dataSource;
    }
   @Bean
    public JdbcTemplate jdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }
}
```

Usage:

- Load the Java configuration in your application:

ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfig.class);
JdbcTemplate jdbcTemplate = context.getBean(JdbcTemplate.class);

Notes:

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

---

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

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

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

How it Works

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

Why is a Connection Pool Needed?
Connection pools are essential for improving the performance, scalability, and resource efficiency of database-driven applications. Here's why:

1. **Performance Improvement**:
    - Establishing a new database connection is an expensive operation, involving network communication, authentication, and resource allocation on both the application and database server.
    - Reusing existing connections from a pool eliminates this overhead, significantly reducing latency for database operations.
2. **Resource Efficiency**:

- Without a connection pool, each request might open a new connection, consuming database resources (memory, threads, sockets) and potentially exhausting the database server's capacity.
- A connection pool limits the number of open connections, ensuring efficient use of resources on both the application and database sides.

3. **Scalability**:
   - In high-concurrency environments (e.g., web applications with many users), creating a new connection for each request can overwhelm the database, leading to failures or slowdowns.
   - Connection pools allow applications to handle multiple requests with a limited number of connections, improving scalability.
4. **Connection Management**:
   - Connection pools handle tasks like maintaining a minimum and maximum number of connections, closing idle connections, and detecting/replacing broken connections.
   - This reduces the complexity of connection management in the application code.
5. **Reliability**:
   - Pools can validate connections before handing them out (e.g., checking if they're still alive) and recover from transient database issues, improving application robustness.

Using DriverManagerDataSource (Basic)

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

Example: DriverManagerDataSource (XML)

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>
   <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
   <property name="username" value="root"/>
   <property name="password" value="password"/>
</bean>
```

Example: DriverManagerDataSource (Java)

```
@Bean
public DataSource dataSource() {
   DriverManagerDataSource dataSource = new DriverManagerDataSource();
   dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
   dataSource.setUrl("jdbc:mysql://localhost:3306/mydb");
   dataSource.setUsername("root");
   dataSource.setPassword("password");
   return dataSource;
}
```

Notes:

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

Using HikariCP (Production-Grade)

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

Maven Dependency:

```
<dependency>
   <groupId>com.zaxxer</groupId>
   <artifactId>HikariCP</artifactId>
   <version>5.1.0</version>
</dependency>
```

Example: HikariCP (XML)

```
<bean id="dataSource" class="com.zaxxer.hikari.HikariDataSource">
   <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>
   <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/mydb"/>
   <property name="username" value="root"/>
   <property name="password" value="password"/>
   <property name="maximumPoolSize" value="10"/>
   <property name="minimumIdle" value="5"/>
</bean>
```

Example: HikariCP (Java)

```
@Bean
public DataSource dataSource() {
   HikariDataSource dataSource = new HikariDataSource();
   dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
   dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/mydb");
   dataSource.setUsername("root");
   dataSource.setPassword("password");
   dataSource.setMaximumPoolSize(10);
   dataSource.setMinimumIdle(5);
   return dataSource;
}
```

Notes:

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

Using Apache DBCP2 (Production-Grade)

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

Maven Dependency:

```
<dependency>
   <groupId>org.apache.commons</groupId>
   <artifactId>commons-dbcp2</artifactId>
   <version>2.12.0</version>
</dependency>
```


Example: Apache DBCP2 (XML)

```
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">
   <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>
   <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
   <property name="username" value="root"/>
   <property name="password" value="password"/>
   <property name="maxTotal" value="10"/>
   <property name="maxIdle" value="5"/>
</bean>
```

Example: Apache DBCP2 (Java)

```
@Bean
public DataSource dataSource() {
   BasicDataSource dataSource = new BasicDataSource();
   dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
   dataSource.setUrl("jdbc:mysql://localhost:3306/mydb");
   dataSource.setUsername("root");
   dataSource.setPassword("password");
   dataSource.setMaxTotal(10);
   dataSource.setMaxIdle(5);
   return dataSource;
}
```

Notes:

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

Summary

Setting up a Spring JDBC environment involves:

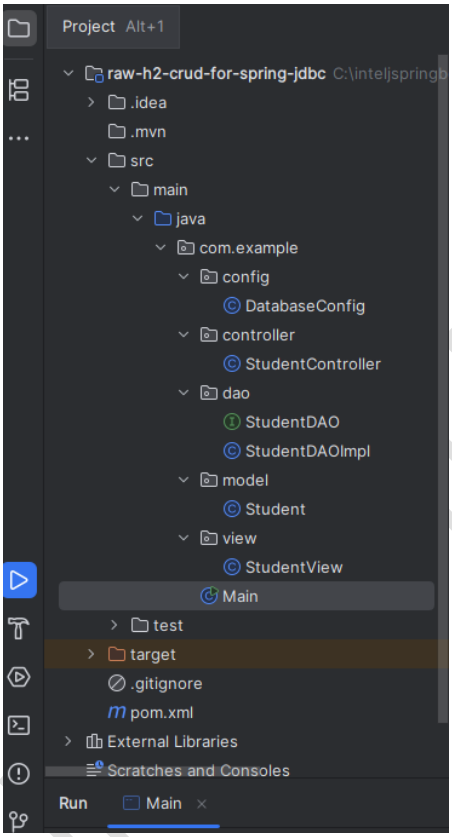
1. **Adding Dependencies**: Include Spring JDBC, a JDBC driver, and optionally a connection pool (HikariCP or Apache DBCP2) in your Maven/Gradle project.
2. **Configuring Spring**:
   - **XML-Based**: Define DataSource and JdbcTemplate beans in an XML file for legacy or externalized configuration.
 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

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

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

Student Database Application:

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



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

```
<dependencies>
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>2.3.230</version>
  </dependency>
</dependencies>
```

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

   @Override
   public Student read(int id) {
      String sql = "SELECT * FROM student WHERE id = ?";
      try (Connection conn = DatabaseConfig.getConnection();
           PreparedStatement pstmt = conn.prepareStatement(sql)) {
         pstmt.setInt(1, id);
         ResultSet rs = pstmt.executeQuery();
         if (rs.next()) {
            return new Student(
               rs.getInt("id"),
               rs.getString("name"),
```


```
                rs.getString("email")
            );
        }
    } catch (SQLException e) {
        System.err.println("Error reading student: " + e.getMessage());
    }
    return null;
}

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

@Override
public void delete(int id) {
    String sql = "DELETE FROM student WHERE id = ?";
    try (Connection conn = DatabaseConfig.getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {
        pstmt.setInt(1, id);
        pstmt.executeUpdate();
    } catch (SQLException e) {
        System.err.println("Error deleting student: " + e.getMessage());
    }
}

@Override
public List<Student> getAll() {
    List<Student> students = new ArrayList<>();
    String sql = "SELECT * FROM student";
    try (Connection conn = DatabaseConfig.getConnection();
         Statement stmt = conn.createStatement();
         ResultSet rs = stmt.executeQuery(sql)) {
        while (rs.next()) {
            students.add(new Student(
                rs.getInt("id"),
                rs.getString("name"),
                rs.getString("email")
            ));
```


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

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

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

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

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);
        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: ");
        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):

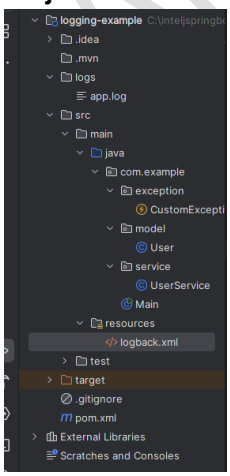
```
<configuration>
  <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n</pattern>
    </encoder>
  </appender>
  <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>logs/app.log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

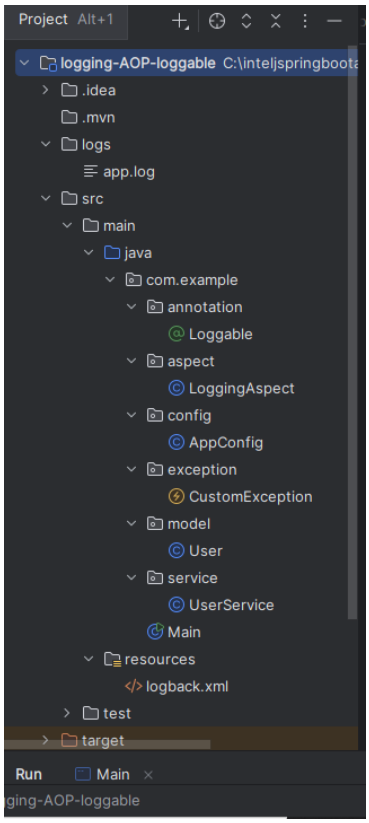
- Console output for immediate feedback
- File output with rolling policy
- Async logging to prevent blocking
- Specific logger for service package with DEBUG level
- Root logger set to INFO level

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>
```


```
    <!-- 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 -> {
```


```
                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)

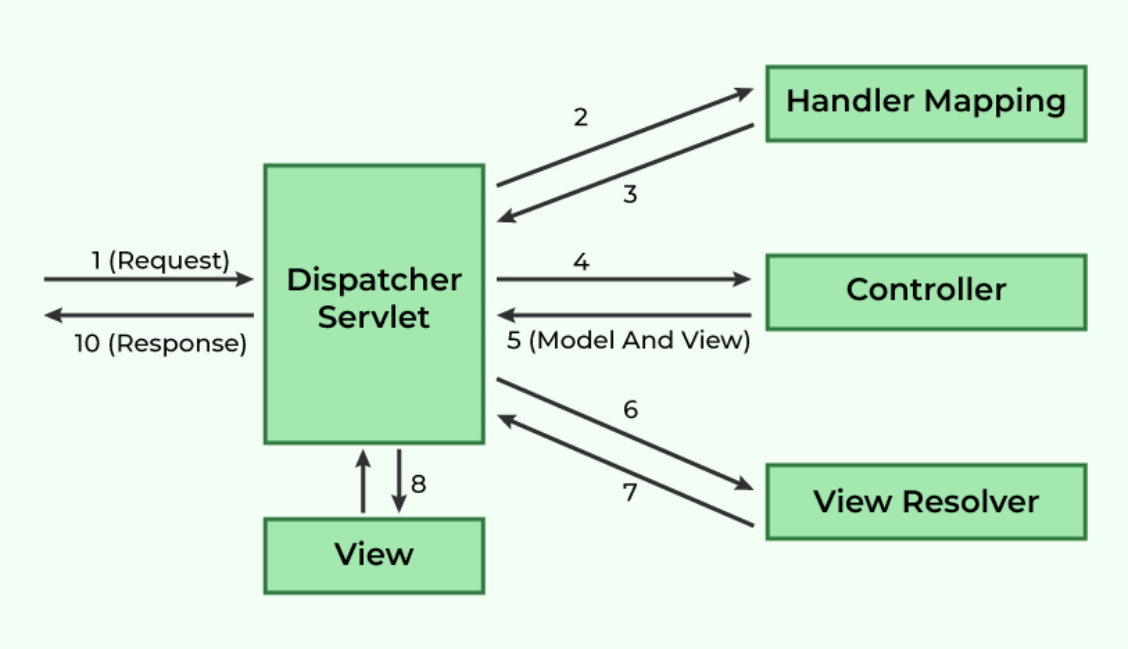
If an error occurs, Spring MVC uses **Exception Handlers** (e.g., @ExceptionHandler or @ControllerAdvice) to handle exceptions and return appropriate responses.

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>
```

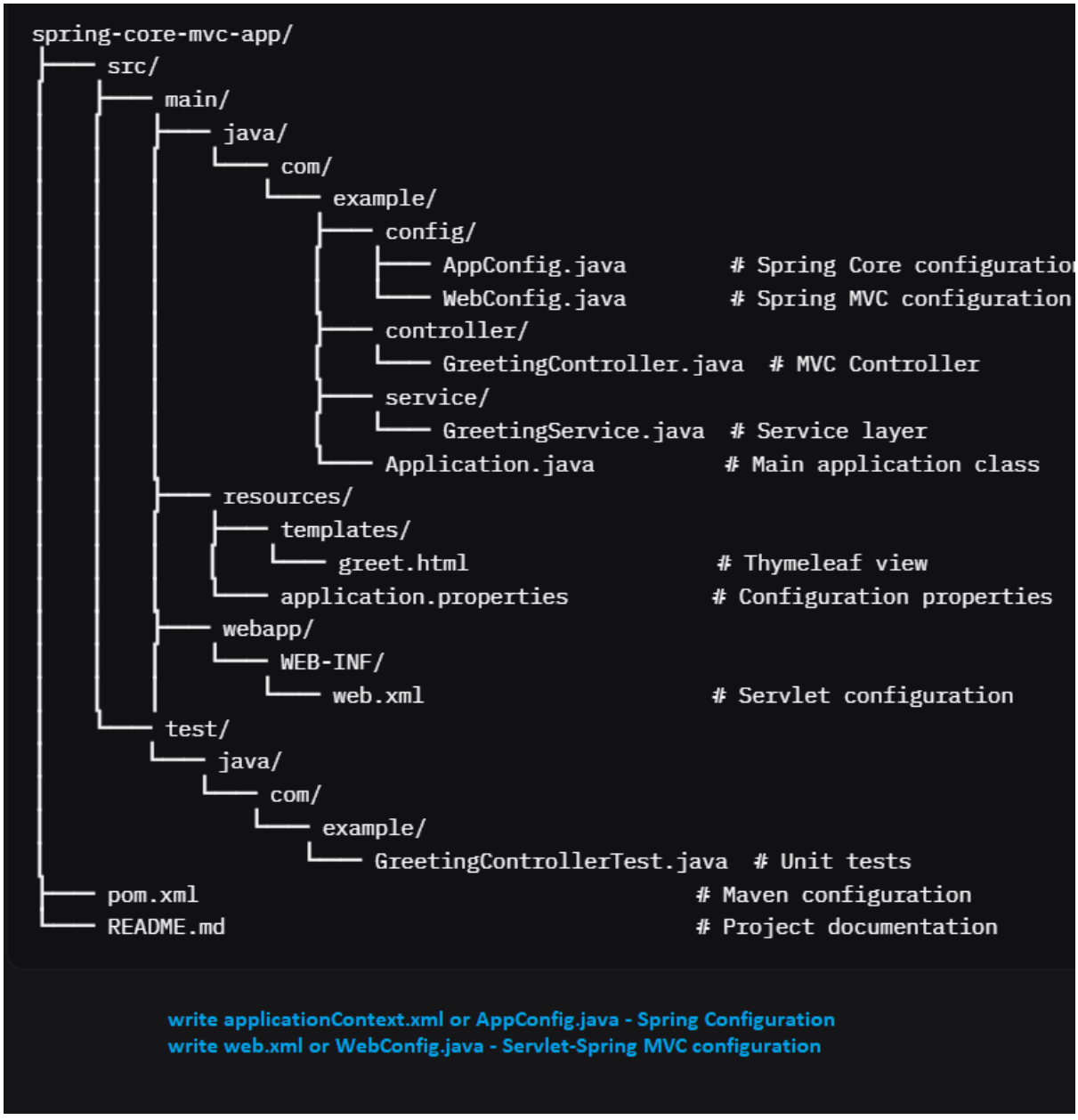
```
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <spring.version>6.1.14</spring.version>
 </properties>

 <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:

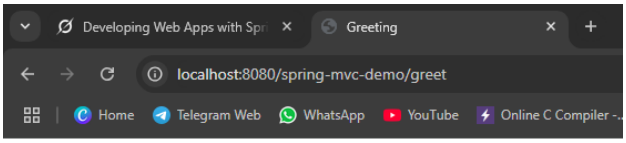
```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
   <title>Greeting</title>
</head>
<body>
   <h1 th:text="${name}">Hello</h1>
</body>
</html>
```

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

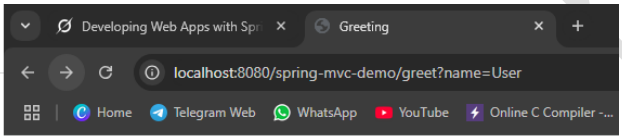
10. Run Application

- 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.

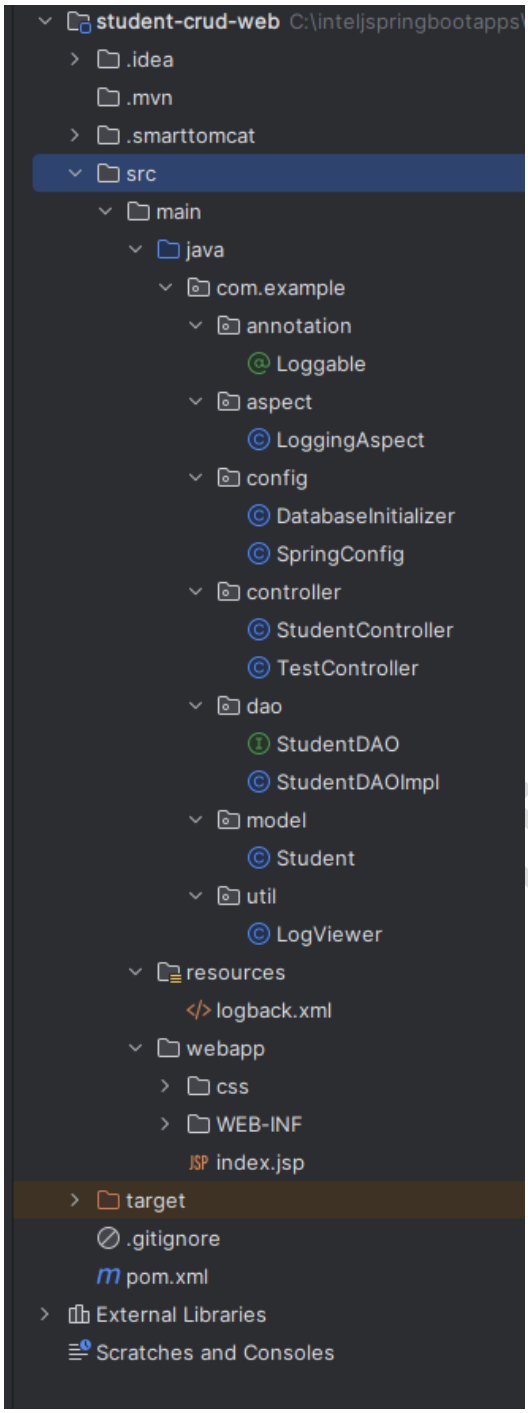
Technology Stack

- **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 depedency
- **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=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 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="${pageContext.request.contextPath}/css/style.css">
</head>
<body>
<div class="container">
    <h1>Welcome to Student Management System</h1>
    <nav>
        <a href="${pageContext.request.contextPath}/students">Manage
Students</a>
        <a
href="${pageContext.request.contextPath}/students/logs">View Logs</a>
    </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="${pageContext.request.contextPath}/css/style.css">
</head>
<body>
<div class="container">
    <h1>Student List</h1>
    <nav>
        <a
href="${pageContext.request.contextPath}/students/create">Create New
Student</a>
        <a
href="${pageContext.request.contextPath}/students/logs">View Logs</a>
        <a href="${pageContext.request.contextPath}/">Home</a>
    </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>
                    <a
href="${pageContext.request.contextPath}/students/${student.id}">View
</a>
                    <a
href="${pageContext.request.contextPath}/students/edit/${student.id}"
>Edit</a>
```


```
                        <a
    href="${pageContext.request.contextPath}/students/delete/${student.id
    }" onclick="return confirm('Are you sure?')">Delete</a>
                    </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="${pageContext.request.contextPath}/css/style.css">
</head>
<body>
<div class="container">
  <h1>Create Student</h1>
  <nav>
    <a href="${pageContext.request.contextPath}/students">Back to List</a>
    <a href="${pageContext.request.contextPath}/students/logs">View Logs</a>
    <a href="${pageContext.request.contextPath}/">Home</a>
  </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/create">
    <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">Create</button>
  </form:form>
</div>
</body>
</html>
```


- **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="${pageContext.request.contextPath}/css/style.css">
</head>
<body>
<div class="container">
    <h1>Edit Student</h1>
    <nav>
        <a href="${pageContext.request.contextPath}/students">Back to
List</a>
        <a
href="${pageContext.request.contextPath}/students/logs">View Logs</a>
        <a href="${pageContext.request.contextPath}/">Home</a>
    </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="${pageContext.request.contextPath}/css/style.css">
</head>
<body>
<div class="container">
    <h1>Student Details</h1>
    <nav>
        <a href="${pageContext.request.contextPath}/students">Back to
```


```
List</a>
        <a
href="${pageContext.request.contextPath}/students/logs">View Logs</a>
        <a href="${pageContext.request.contextPath}/">Home</a>
    </nav>
    <c:if test="${not empty error}">
        <p class="error">${error}</p>
    </c:if>
    <c:if test="${not empty student}">
        <table>
            <tr>
                <th>ID</th>
                <td>${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="${pageContext.request.contextPath}/css/style.css"
rel="stylesheet"/>
</head>
<body>
<div class="container">
    <h1>Application Logs</h1>
    <nav>
        <a href="${pageContext.request.contextPath}/students">Back to
List</a>
        <a href="${pageContext.request.contextPath}/">Home</a>
    </nav>
    <c:choose>
        <c:when test="${logs.startsWith('Unable to')}">
            <p class="error"><c:out value="${logs}" /></p>
        </c:when>
        <c:otherwise>
            <pre><c:out value="${logs}" /></pre>
        </c:otherwise>
    </c:choose>
</div>
</body>
</html>
```


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

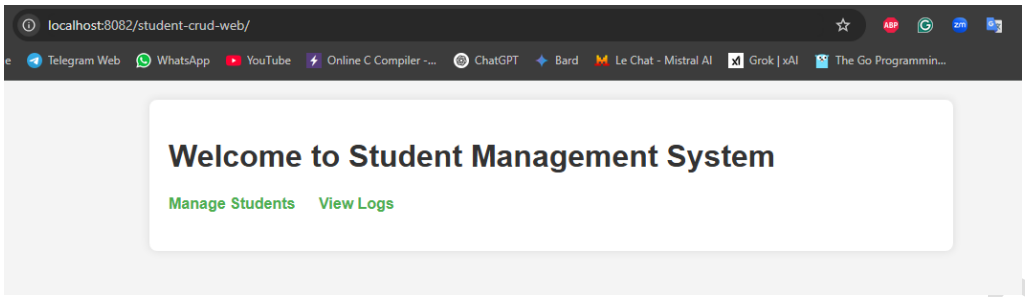
```
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 20px;
    background-color: #f4f4f4;
}
h1, h2 {
    color: #333;
}
.container {
    max-width: 800px;
    margin: 0 auto;
    background: #fff;
    padding: 20px;
    border-radius: 8px;
    box-shadow: 0 0 10px rgba(0,0,0,0.1);
}
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;
}
```

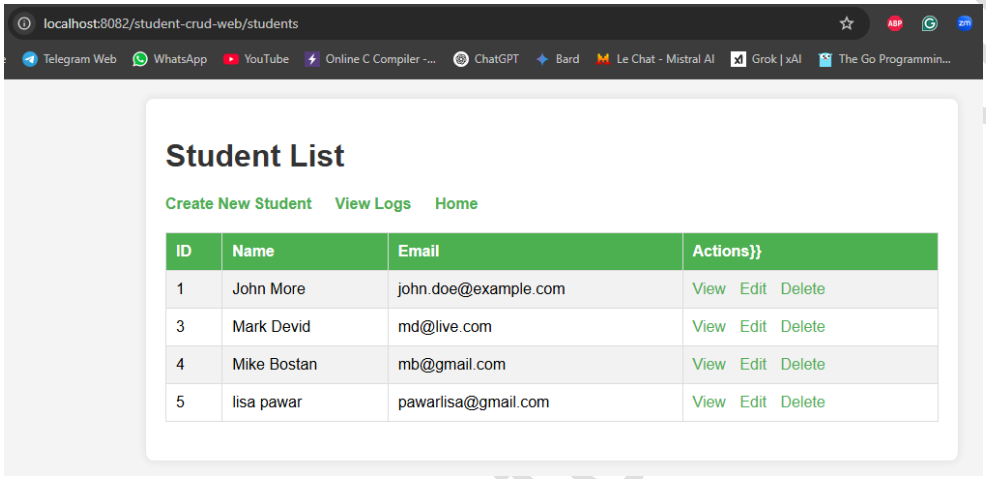

```
    button:hover {
        background-color: #45a049;
    }
    .error {
        color: red;
        font-weight: bold;
    }
    pre {
        background: #f8f8f8;
        padding: 10px;
        border: 1px solid #ddd;
        border-radius: 4px;
        max-height: 400px;
        overflow-y: auto;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

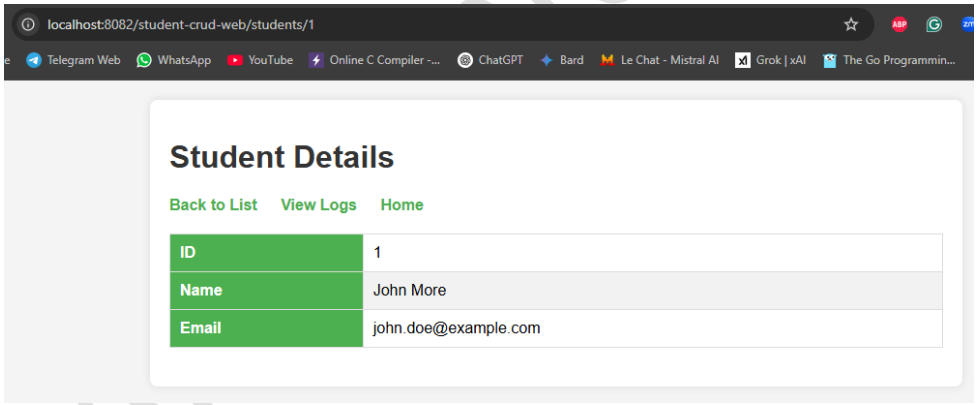
- **logback.xml**: Configures Logback to write logs to student-crud-web/logs/student-crud.log.

```
<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>
```



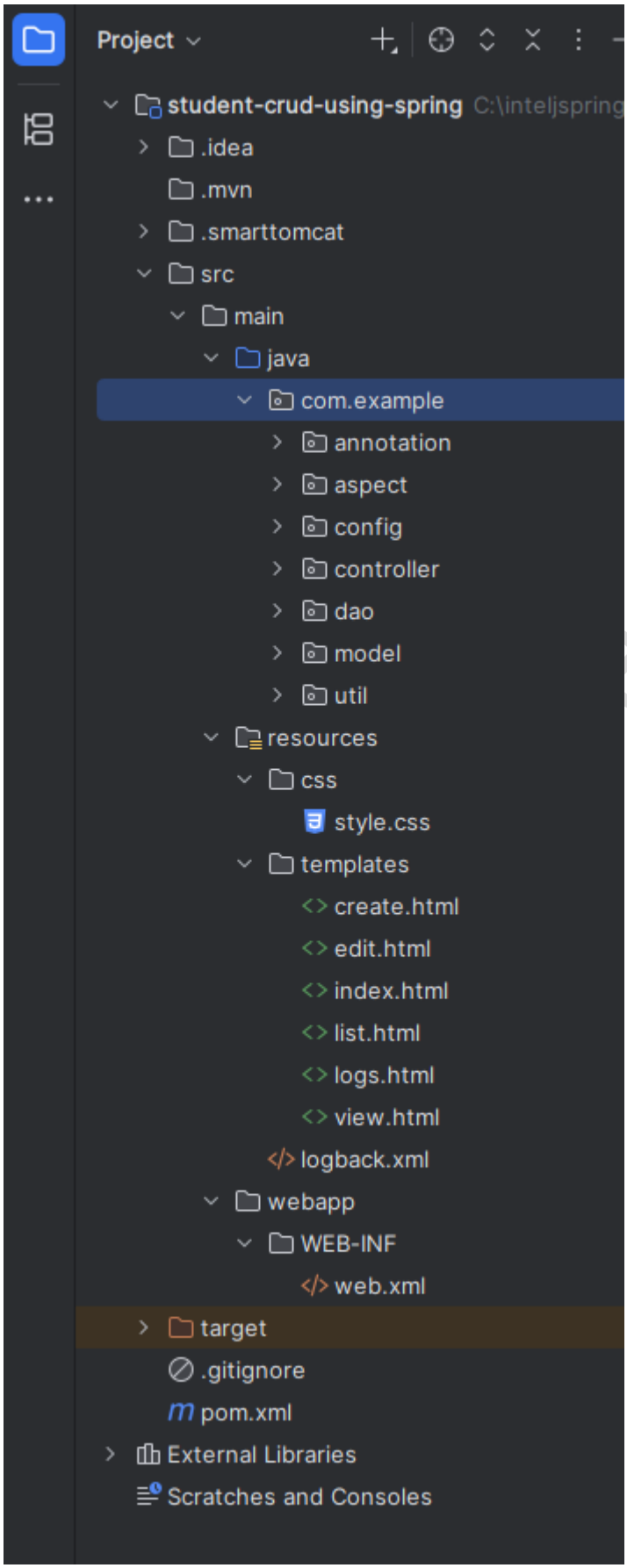




Access URLs:

- **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("classpat
h:/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>
```


```
            <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>
```


```
            <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;
    }
```

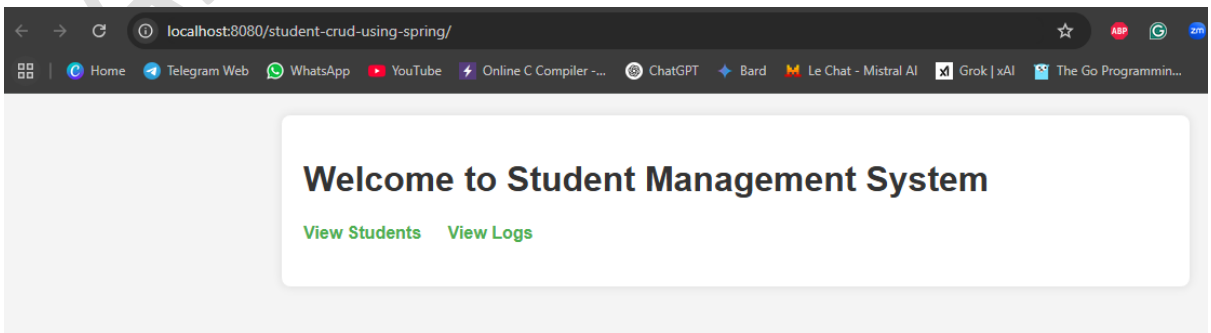

```
th {
    background-color: #4CAF50;
    color: white;
}
tr:nth-child(even) {
    background-color: #f2f2f2;
}
a {
    color: #4CAF50;
    text-decoration: none;
    margin-right: 10px;
}
a:hover {
    text-decoration: underline;
}
form {
    display: flex;
    flex-direction: column;
    gap: 10px;
}
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;
}
```

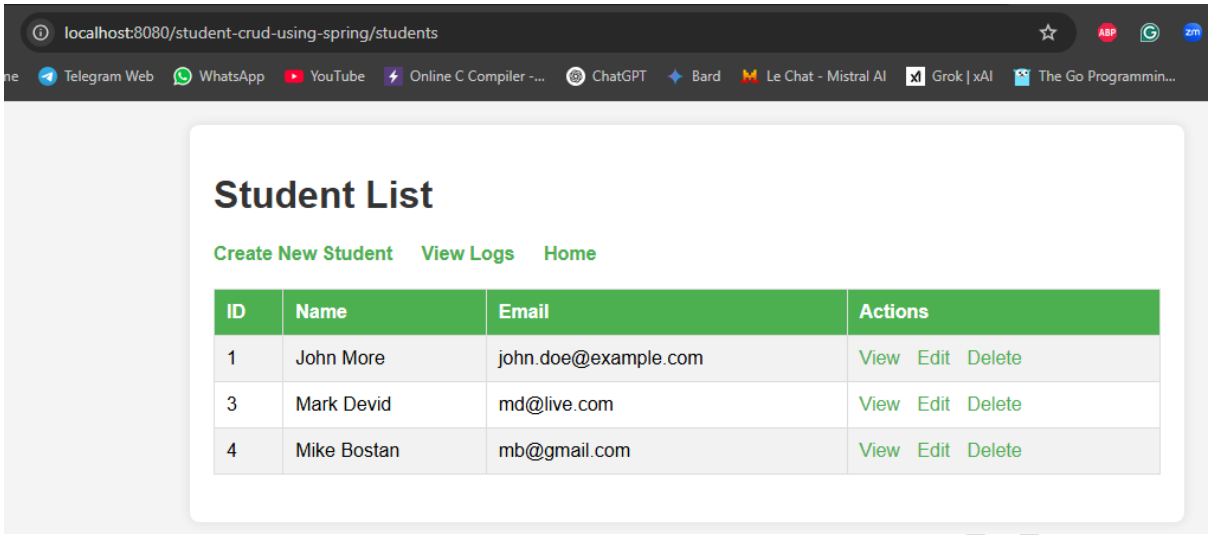

- **logback.xml**

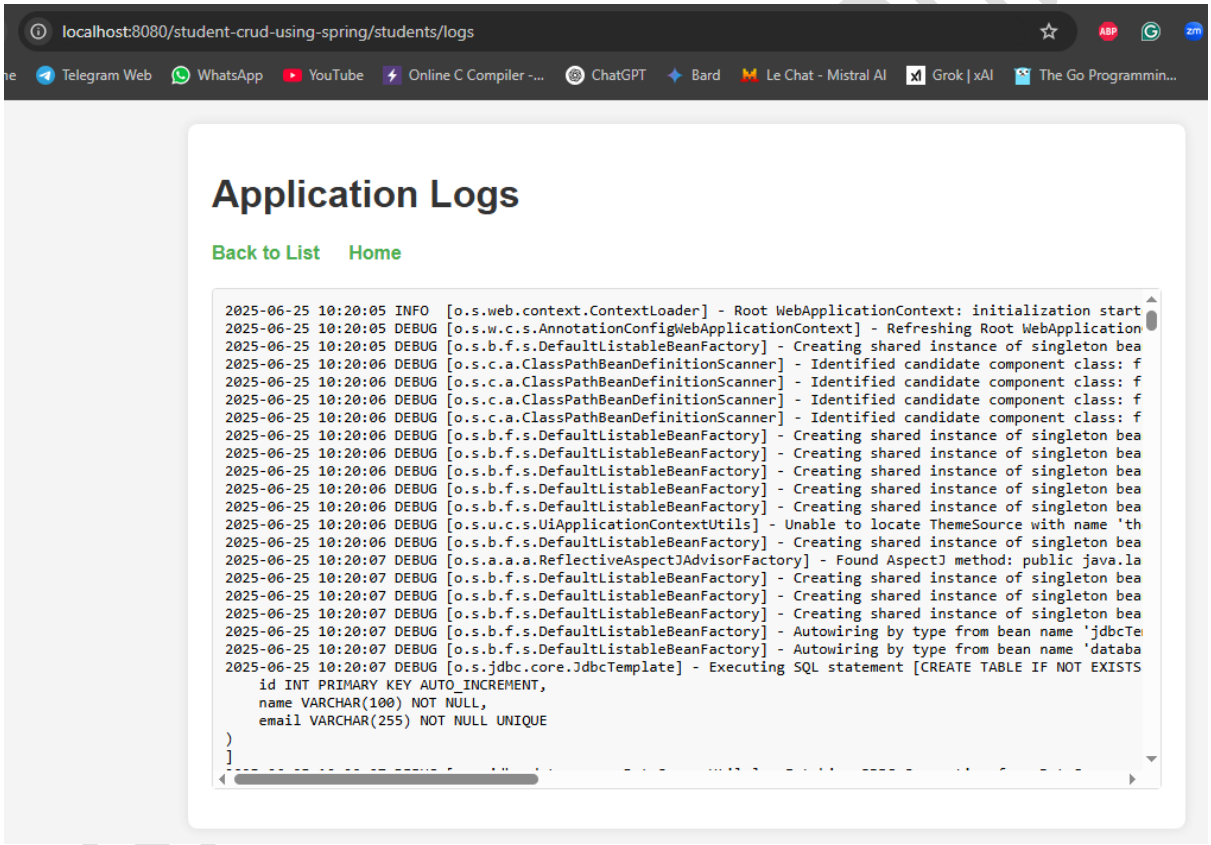
```
<configuration>
    <property name="LOG_DIR" value="${user.dir}/logs" />
    <appender name="CONSOLE"
class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level [%logger{36}] -
%msg%n</pattern>
        </encoder>
    </appender>
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>${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>
```

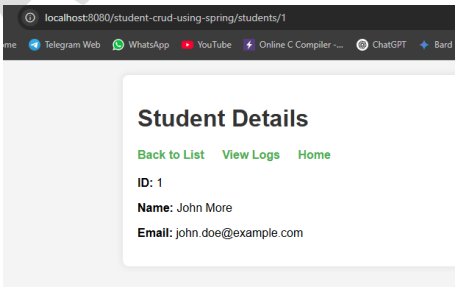
Accessible URLs

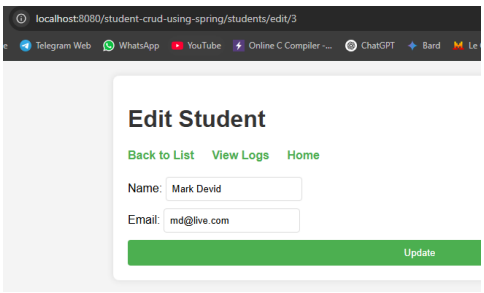
- http://localhost:8080/student-crud-web/ - Displays the welcome page (index.html).
- http://localhost:8080/student-crud-web/students - Lists all students (list.html).
- http://localhost:8080/student-crud-web/students/test - Test endpoint, displays an empty student list (list.html).
- http://localhost:8080/student-crud-web/students/create - 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
Telegram Web
WhatsApp
YouTube
Online C Compiler -...
ChatGPT
Bard
Le Chat - Mistral AI
Grok | xAI
The Go Programmin...
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 bea
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 bea
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bea
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bea
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bea
2025-06-25 10:20:06 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bea
2025-06-25 10:20:06 DEBUG [o.s.u.c.s.UiApplicationContextUtils] - 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 bea
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 bea
2025-06-25 10:20:07 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bea
2025-06-25 10:20:07 DEBUG [o.s.b.f.s.DefaultListableBeanFactory] - Creating shared instance of singleton bea
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.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/3
Edit Student
Back to List
View Logs
Home
Name: Mark Devid
Email: md@live.com
Update