DAO (Data Access Object) Spring project

What is DAO in Spring?

DAO (**Data Access Object**) is a **design pattern** that separates the **persistence logic** (database operations) from the **business logic** (service/application layer).

In **Spring JDBC**, DAO classes typically use **JdbcTemplate** (or other ORM tools like Hibernate) to perform database operations.

Why DAO?

- Keeps code **modular & reusable**
- Makes unit testing easier
- Clean separation of **database logic** and **application logic**

Structure of a Spring DAO Project

A typical Spring JDBC DAO project has the following layers:

1. **Entity Layer (Model Classes)**
   - Java classes mapped to database tables (POJOs).
   - Example: `Department`, `Employee`.
2. **DAO Layer**
   - Contains classes with methods for CRUD operations.
   - Uses **JdbcTemplate** for DB access.
   - Example: `DepartmentDAO`, `EmployeeDAO`.
3. **Service Layer (Optional, Business Logic)**
   - Calls DAOs to implement business rules.
   - Example: `EmployeeService` might call both `EmployeeDAO` and `DepartmentDAO`.
4. **Configuration Layer**
   - Provides DB connection (`DataSource`) and `JdbcTemplate` beans.
   - Can be done using **Java config** (`@Configuration`) or XML config.
5. **Main (Runner / Application Layer)**
   - Bootstraps Spring, calls services/DAOs.

Spring JDBC Template project with two entities (Employee & Department)

Project Overview

This project is a **Spring JDBC Template-based CRUD application** in Java that demonstrates how to work with relational databases using **Spring's JdbcTemplate API** instead of writing raw JDBC code.

We use **two entities**:

- **Department** → Each department has `id`, `name`.
- **Employee** → Each employee has `id`, `name`, `salary`, and a reference to a department (`department_id`).

Components Involved

1. Dependencies (Maven)

For Oracle (`ojdbc8`) or H2 (for testing), you need:

- `spring-context` → Core Spring container
- `spring-jdbc` → For `JdbcTemplate`
- `spring-tx` → For transaction management
- `ojdbc8` (for Oracle) or `h2` (for testing)

---

2. Configuration

We define a `DataSource` bean (database connection) and a `JdbcTemplate` bean.

For **Oracle with ojdbc8**:

```
@Configuration
public class DataSourceConfig {
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource ds = new
DriverManagerDataSource();
        ds.setDriverClassName("oracle.jdbc.OracleDriver");
        ds.setUrl("jdbc:oracle:thin:@localhost:1521:xe"); //
adjust SID/service name
```

```
        ds.setUsername("your_username");
        ds.setPassword("your_password");
        return ds;
    }

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

---

3. Entities

```
public class Department {
    private int id;
    private String name;
    // getters, setters, toString
}

public class Employee {
    private int id;
    private String name;
    private double salary;
    private int departmentId; // Foreign key
    // getters, setters, toString
}
```

---

4. DAO Layer (Data Access Objects)

Here we use **JdbcTemplate** to execute SQL queries.

```
@Repository
public class DepartmentDAO {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    public void save(Department dept) {
        String sql = "INSERT INTO department (id, name) VALUES
(?, ?)";
        jdbcTemplate.update(sql, dept.getId(), dept.getName());
    }
}

@Repository
public class EmployeeDAO {
```

```
    @Autowired
    private JdbcTemplate jdbcTemplate;

    public void save(Employee emp) {
        String sql = "INSERT INTO employee (id, name, salary,
department_id) VALUES (?, ?, ?, ?)";
        jdbcTemplate.update(sql, emp.getId(), emp.getName(),
emp.getSalary(), emp.getDepartmentId());
    }
}
```

---

5. Main Application

```
public class App {
    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(DataSourceConfig.class);

        DepartmentDAO deptDao =
context.getBean(DepartmentDAO.class);
        EmployeeDAO empDao = context.getBean(EmployeeDAO.class);

        // Insert Department
        Department dept = new Department();
        dept.setId(101);
        dept.setName("IT");
        deptDao.save(dept);

        // Insert Employee
        Employee emp = new Employee();
        emp.setId(1);
        emp.setName("Ravi");
        emp.setSalary(50000);
        emp.setDepartmentId(101);
        empDao.save(emp);

        System.out.println("Data inserted successfully!");
    }
}
```

Database Schema (Oracle DDL)

Before running, you must create the tables manually in Oracle:

```
CREATE TABLE department (
```

```
    id NUMBER PRIMARY KEY,
    name VARCHAR2(100) NOT NULL
);

CREATE TABLE employee (
    id NUMBER PRIMARY KEY,
    name VARCHAR2(100) NOT NULL,
    salary NUMBER(10,2),
    department_id NUMBER,
    CONSTRAINT fk_dept FOREIGN KEY (department_id) REFERENCES
department(id)
);
```

Flow of Execution

1. Spring loads `DataSource` and `JdbcTemplate`.
2. The DAO classes use `JdbcTemplate.update()` to execute SQL queries.
3. In `App`, we create a Department and an Employee.
4. The `department` record gets inserted into Oracle first.
5. The `employee` record gets inserted with a foreign key linking to the department.

This project shows how to use **Spring JDBC Template with Oracle (ojdbc8)** to perform CRUD operations on two related entities (`Department`, `Employee`) in a **clean, Spring-managed way**, avoiding boilerplate JDBC code.

Would you like me to also add **read (SELECT), update, and delete methods** in DAO so it becomes a full-fledged CRUD project instead of just insert-only?