JDBC Best Practices

Best-practice JDBC (MySQL) CRUD example for **Employee records**, designed with clean **OOP layering**:

- **Model** (`Employee`)
- **DAO interface + JDBC implementation** (`EmployeeDao`, `JdbcEmployeeDao`)
- **Service layer** (`EmployeeService`) for validation + business rules
- **Connection factory** (`DbConnectionFactory`) using properties/env
- **Custom exceptions** (`DaoException`, `NotFoundException`, `ValidationException`)
- **Try-with-resources**, **PreparedStatement**, **transaction example**, **safe mapping**

You can copy this into a plain Java project.

---

0) MySQL table (DDL)

```sql
CREATE DATABASE IF NOT EXISTS company_db;
USE company_db;

CREATE TABLE IF NOT EXISTS employees (
  id BIGINT PRIMARY KEY AUTO_INCREMENT,
  emp_code VARCHAR(20) NOT NULL UNIQUE,
  full_name VARCHAR(100) NOT NULL,
  email VARCHAR(120) NOT NULL UNIQUE,
  department VARCHAR(60) NOT NULL,
  salary DECIMAL(12,2) NOT NULL,
  created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
);

CREATE INDEX idx_employees_department ON employees(department);
```

---

1) Project structure (recommended)

```
com.company.employee
  ├─ model
  │    └─ Employee.java
  ├─ dao
  │    ├─ EmployeeDao.java
  │    ├─ JdbcEmployeeDao.java
  │    └─ DaoException.java
  ├─ service
  │    ├─ EmployeeService.java
  │    ├─ ValidationException.java
  │    └─ NotFoundException.java
  ├─ db
  │    └─ DbConnectionFactory.java
  └─ Main.java
```

---

2) Dependency (MySQL JDBC driver)

Maven:

```xml
<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
  <version>9.0.0</version>
</dependency>
```

---

3) DB connection factory (best practices)

- One place to manage JDBC URL/user/password
- Keeps credentials out of code (use env vars or system properties)

`DbConnectionFactory.java`

```java
package com.company.employee.db;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Objects;

public final class DbConnectionFactory {
    private final String url;
    private final String user;
    private final String password;

    public DbConnectionFactory(String url, String user, String
password) {
        this.url = Objects.requireNonNull(url);
        this.user = Objects.requireNonNull(user);
        this.password = Objects.requireNonNull(password);
    }

    public Connection getConnection() throws SQLException {
        // For production: use a connection pool (HikariCP). This is
fine for learning/interviews.
        return DriverManager.getConnection(url, user, password);
    }

    public static DbConnectionFactory fromEnv() {
        // Example env vars:
        //
DB_URL="jdbc:mysql://localhost:3306/company_db?useSSL=false&serverTi
mezone=UTC"
        // DB_USER="root"
        // DB_PASSWORD="root123"
        String url = System.getenv().getOrDefault("DB_URL",

"jdbc:mysql://localhost:3306/company_db?useSSL=false&serverTimezone=
UTC");
        String user = System.getenv().getOrDefault("DB_USER",
"root");
        String pass = System.getenv().getOrDefault("DB_PASSWORD",
"");
        return new DbConnectionFactory(url, user, pass);
    }
}
```

---

4) Model (Employee)

`Employee.java`

```java
package com.company.employee.model;

import java.math.BigDecimal;
import java.time.Instant;
import java.util.Objects;

public final class Employee {
    private final Long id; // null for new employees
    private final String empCode;
    private final String fullName;
    private final String email;
    private final String department;
    private final BigDecimal salary;
    private final Instant createdAt; // optional in app layer
    private final Instant updatedAt;

    private Employee(Builder b) {
        this.id = b.id;
        this.empCode = Objects.requireNonNull(b.empCode);
        this.fullName = Objects.requireNonNull(b.fullName);
        this.email = Objects.requireNonNull(b.email);
        this.department = Objects.requireNonNull(b.department);
        this.salary = Objects.requireNonNull(b.salary);
        this.createdAt = b.createdAt;
        this.updatedAt = b.updatedAt;
    }

    public Long getId() { return id; }
    public String getEmpCode() { return empCode; }
    public String getFullName() { return fullName; }
    public String getEmail() { return email; }
    public String getDepartment() { return department; }
    public BigDecimal getSalary() { return salary; }
    public Instant getCreatedAt() { return createdAt; }
    public Instant getUpdatedAt() { return updatedAt; }

    public Builder toBuilder() {
        return new Builder()
```

```
.id(id).empCode(empCode).fullName(fullName).email(email)
                .department(department).salary(salary)
                .createdAt(createdAt).updatedAt(updatedAt);
    }

    public static Builder builder() { return new Builder(); }

    public static final class Builder {
        private Long id;
        private String empCode;
        private String fullName;
        private String email;
        private String department;
        private BigDecimal salary;
        private Instant createdAt;
        private Instant updatedAt;

        public Builder id(Long id) { this.id = id; return this; }
        public Builder empCode(String empCode) { this.empCode =
empCode; return this; }
        public Builder fullName(String fullName) { this.fullName =
fullName; return this; }
        public Builder email(String email) { this.email = email;
return this; }
        public Builder department(String department) {
this.department = department; return this; }
        public Builder salary(BigDecimal salary) { this.salary =
salary; return this; }
        public Builder createdAt(Instant createdAt) { this.createdAt
= createdAt; return this; }
        public Builder updatedAt(Instant updatedAt) { this.updatedAt
= updatedAt; return this; }

        public Employee build() { return new Employee(this); }
    }

    @Override
    public String toString() {
        return "Employee{id=" + id + ", empCode='" + empCode + "',
fullName='" + fullName +
                "', email='" + email + "', department='" +
department + "', salary=" + salary + "}";
    }
}
```

Why this design is robust:

- Immutable objects reduce accidental state bugs.
- Builder makes object creation readable and safe.

---

5) DAO contract + exceptions

`EmployeeDao.java`

```
package com.company.employee.dao;

import com.company.employee.model.Employee;

import java.util.List;
import java.util.Optional;

public interface EmployeeDao {
    Employee create(Employee employee);
    Optional<Employee> findById(long id);
    Optional<Employee> findByEmpCode(String empCode);
    List<Employee> findAll(int limit, int offset);
    List<Employee> findByDepartment(String department, int limit,
int offset);
    Employee update(Employee employee);          // by id
    boolean deleteById(long id);
}
```

`DaoException.java`

```
package com.company.employee.dao;

public class DaoException extends RuntimeException {
    public DaoException(String message, Throwable cause) {
super(message, cause); }
}
```

---

6) JDBC DAO implementation (CRUD with best practices)

Key best practices used:

- `PreparedStatement` (prevents SQL injection)
- `try-with-resources` (always closes resources)
- `RETURN_GENERATED_KEYS` (proper insert id)
- Centralized mapping `mapRow(ResultSet)`
- Clear SQL constants

`JdbcEmployeeDao.java`

```
package com.company.employee.dao;

import com.company.employee.db.DbConnectionFactory;
import com.company.employee.model.Employee;

import java.math.BigDecimal;
import java.sql.*;
import java.time.Instant;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

public final class JdbcEmployeeDao implements EmployeeDao {

    private static final String INSERT_SQL =
            "INSERT INTO employees (emp_code, full_name, email,
department, salary) VALUES (?, ?, ?, ?, ?)";

    private static final String SELECT_BY_ID_SQL =
            "SELECT id, emp_code, full_name, email, department,
salary, created_at, updated_at FROM employees WHERE id=?";

    private static final String SELECT_BY_CODE_SQL =
            "SELECT id, emp_code, full_name, email, department,
salary, created_at, updated_at FROM employees WHERE emp_code=?";

    private static final String SELECT_ALL_SQL =
            "SELECT id, emp_code, full_name, email, department,
salary, created_at, updated_at FROM employees ORDER BY id LIMIT ?
OFFSET ?";

    private static final String SELECT_BY_DEPT_SQL =
            "SELECT id, emp_code, full_name, email, department,
salary, created_at, updated_at FROM employees WHERE department=?
ORDER BY id LIMIT ? OFFSET ?";

    private static final String UPDATE_SQL =
            "UPDATE employees SET full_name=?, email=?,
department=?, salary=? WHERE id=?";

    private static final String DELETE_SQL =
            "DELETE FROM employees WHERE id=?";

    private final DbConnectionFactory connectionFactory;

    public JdbcEmployeeDao(DbConnectionFactory connectionFactory) {
        this.connectionFactory = connectionFactory;
    }

    @Override
    public Employee create(Employee employee) {
        try (Connection con = connectionFactory.getConnection();
             PreparedStatement ps = con.prepareStatement(INSERT_SQL,
Statement.RETURN_GENERATED_KEYS)) {

            ps.setString(1, employee.getEmpCode());
            ps.setString(2, employee.getFullName());
            ps.setString(3, employee.getEmail());
            ps.setString(4, employee.getDepartment());
            ps.setBigDecimal(5, employee.getSalary());
            int affected = ps.executeUpdate();
            if (affected != 1) {
                throw new DaoException("Insert failed: expected 1
row, affected=" + affected, null);
            }

            try (ResultSet keys = ps.getGeneratedKeys()) {
                if (!keys.next()) {
                    throw new DaoException("Insert failed: no
generated key returned", null);
                }
                long id = keys.getLong(1);

                // Fetch the inserted row (ensures timestamps match
DB)
                return findById(id).orElseThrow(() -> new
DaoException("Inserted row not found for id=" + id, null));
            }

        } catch (SQLException e) {
            throw new DaoException("Error creating employee", e);
        }
    }

    @Override
    public Optional<Employee> findById(long id) {
        try (Connection con = connectionFactory.getConnection();
             PreparedStatement ps =
con.prepareStatement(SELECT_BY_ID_SQL)) {

            ps.setLong(1, id);
            try (ResultSet rs = ps.executeQuery()) {
                if (!rs.next()) return Optional.empty();
                return Optional.of(mapRow(rs));
            }

        } catch (SQLException e) {
            throw new DaoException("Error finding employee by id=" +
id, e);
        }
    }

    @Override
    public Optional<Employee> findByEmpCode(String empCode) {
        try (Connection con = connectionFactory.getConnection();
             PreparedStatement ps =
con.prepareStatement(SELECT_BY_CODE_SQL)) {

            ps.setString(1, empCode);
            try (ResultSet rs = ps.executeQuery()) {
                if (!rs.next()) return Optional.empty();
                return Optional.of(mapRow(rs));
            }

        } catch (SQLException e) {
            throw new DaoException("Error finding employee by
empCode=" + empCode, e);
        }
    }

    @Override
    public List<Employee> findAll(int limit, int offset) {
        try (Connection con = connectionFactory.getConnection();
             PreparedStatement ps =
con.prepareStatement(SELECT_ALL_SQL)) {

            ps.setInt(1, limit);
            ps.setInt(2, offset);

            try (ResultSet rs = ps.executeQuery()) {
                List<Employee> out = new ArrayList<>();
                while (rs.next()) out.add(mapRow(rs));
                return out;
            }

        } catch (SQLException e) {
            throw new DaoException("Error listing employees", e);
        }
    }

    @Override
    public List<Employee> findByDepartment(String department, int
limit, int offset) {
        try (Connection con = connectionFactory.getConnection();
             PreparedStatement ps =
con.prepareStatement(SELECT_BY_DEPT_SQL)) {

            ps.setString(1, department);
            ps.setInt(2, limit);
            ps.setInt(3, offset);

            try (ResultSet rs = ps.executeQuery()) {
                List<Employee> out = new ArrayList<>();
                while (rs.next()) out.add(mapRow(rs));
                return out;
            }

        } catch (SQLException e) {
            throw new DaoException("Error listing employees by
department=" + department, e);
        }
    }

    @Override
    public Employee update(Employee employee) {
        if (employee.getId() == null) {
            throw new DaoException("Update requires employee.id (was
null)", null);
        }

        try (Connection con = connectionFactory.getConnection();
             PreparedStatement ps =
con.prepareStatement(UPDATE_SQL)) {

            ps.setString(1, employee.getFullName());
            ps.setString(2, employee.getEmail());
            ps.setString(3, employee.getDepartment());
            ps.setBigDecimal(4, employee.getSalary());
            ps.setLong(5, employee.getId());

            int affected = ps.executeUpdate();
            if (affected != 1) {
                throw new DaoException("Update failed: expected 1
row, affected=" + affected, null);
            }

            return findById(employee.getId())
                    .orElseThrow(() -> new DaoException("Updated row
not found for id=" + employee.getId(), null));

        } catch (SQLException e) {
            throw new DaoException("Error updating employee id=" +
employee.getId(), e);
        }
    }

    @Override
    public boolean deleteById(long id) {
        try (Connection con = connectionFactory.getConnection();
             PreparedStatement ps =
con.prepareStatement(DELETE_SQL)) {

            ps.setLong(1, id);
            int affected = ps.executeUpdate();
            return affected == 1;

        } catch (SQLException e) {
            throw new DaoException("Error deleting employee id=" +
id, e);
        }
    }

    private Employee mapRow(ResultSet rs) throws SQLException {
        long id = rs.getLong("id");
        String empCode = rs.getString("emp_code");
        String fullName = rs.getString("full_name");
        String email = rs.getString("email");
        String department = rs.getString("department");
        BigDecimal salary = rs.getBigDecimal("salary");

        Timestamp created = rs.getTimestamp("created_at");
        Timestamp updated = rs.getTimestamp("updated_at");
        return Employee.builder()
                .id(id)
                .empCode(empCode)
                .fullName(fullName)
                .email(email)
                .department(department)
                .salary(salary)
                .createdAt(created != null ? created.toInstant() :
null)
                .updatedAt(updated != null ? updated.toInstant() :
null)
                .build();
    }
}
```

---

7) Service layer (validation + business rules)

Why service layer?

- DAO should focus on DB operations
- Service should enforce business rules and validations

ValidationException.java

```
package com.company.employee.service;

public class ValidationException extends RuntimeException {
    public ValidationException(String message) { super(message); }
}
```

NotFoundException.java

```
package com.company.employee.service;

public class NotFoundException extends RuntimeException {
    public NotFoundException(String message) { super(message); }
}
```

EmployeeService.java

```
package com.company.employee.service;

import com.company.employee.dao.EmployeeDao;
import com.company.employee.model.Employee;

import java.math.BigDecimal;
import java.util.List;
import java.util.Objects;

public final class EmployeeService {
    private final EmployeeDao dao;

    public EmployeeService(EmployeeDao dao) {
        this.dao = Objects.requireNonNull(dao);
    }

    public Employee createEmployee(Employee e) {
        validateForCreate(e);

        // Example robustness: prevent duplicate empCode early (DB
unique will also protect you)
        dao.findByEmpCode(e.getEmpCode()).ifPresent(existing -> {
            throw new ValidationException("Employee code already
exists: " + e.getEmpCode());
        });

        return dao.create(e);
    }

    public Employee getEmployee(long id) {
        return dao.findById(id).orElseThrow(() -> new
NotFoundException("Employee not found: id=" + id));
    }

    public List<Employee> listEmployees(int limit, int offset) {
        if (limit <= 0 || limit > 1000) throw new
ValidationException("limit must be 1..1000");
        if (offset < 0) throw new ValidationException("offset must
be >= 0");
        return dao.findAll(limit, offset);
    }

    public Employee updateEmployee(Employee e) {
        validateForUpdate(e);

        // Ensure exists
        getEmployee(e.getId());

        return dao.update(e);
    }

    public void deleteEmployee(long id) {
        boolean deleted = dao.deleteById(id);
        if (!deleted) throw new NotFoundException("Employee not
found to delete: id=" + id);
    }

    private void validateForCreate(Employee e) {
        Objects.requireNonNull(e, "employee required");
        if (e.getId() != null) throw new ValidationException("New
employee must not have id");
        validateCommon(e);
    }

    private void validateForUpdate(Employee e) {
        Objects.requireNonNull(e, "employee required");
        if (e.getId() == null) throw new ValidationException("Update
requires id");
        validateCommon(e);
    }

    private void validateCommon(Employee e) {
        if (isBlank(e.getEmpCode())) throw new
ValidationException("empCode required");
        if (e.getEmpCode().length() > 20) throw new
ValidationException("empCode max length 20");
        if (isBlank(e.getFullName())) throw new
ValidationException("fullName required");
        if (isBlank(e.getEmail()) || !e.getEmail().contains("@"))
throw new ValidationException("valid email required");
```

```
        if (isBlank(e.getDepartment())) throw new
ValidationException("department required");
        if (e.getSalary() == null ||
e.getSalary().compareTo(BigDecimal.ZERO) <= 0) throw new
ValidationException("salary must be > 0");
    }

    private boolean isBlank(String s) {
        return s == null || s.trim().isEmpty();
    }
}
```

---

8) Main demo (CRUD run)

`Main.java`

```
package com.company.employee;

import com.company.employee.dao.EmployeeDao;
import com.company.employee.dao.JdbcEmployeeDao;
import com.company.employee.db.DbConnectionFactory;
import com.company.employee.model.Employee;
import com.company.employee.service.EmployeeService;

import java.math.BigDecimal;

public class Main {
    public static void main(String[] args) {
        DbConnectionFactory factory = DbConnectionFactory.fromEnv();
        EmployeeDao dao = new JdbcEmployeeDao(factory);
        EmployeeService service = new EmployeeService(dao);

        // CREATE
        Employee created = service.createEmployee(Employee.builder()
                .empCode("EMP-1001")
                .fullName("Aarav Sharma")
                .email("aarav.sharma@example.com")
                .department("Engineering")
                .salary(new BigDecimal("120000.00"))
                .build());
        System.out.println("Created: " + created);

        // READ
        Employee fetched = service.getEmployee(created.getId());
        System.out.println("Fetched: " + fetched);

        // UPDATE
        Employee updated =
service.updateEmployee(fetched.toBuilder()
                .department("Platform Engineering")
                .salary(new BigDecimal("135000.00"))
                .build());
        System.out.println("Updated: " + updated);

        // LIST
        System.out.println("List:");
        service.listEmployees(10, 0).forEach(System.out::println);

        // DELETE
        service.deleteEmployee(updated.getId());
        System.out.println("Deleted id=" + updated.getId());
    }
}
```

---

9) Robustness best practices used (what makes this “good”)

✅ **PreparedStatement** prevents SQL injection
✅ **try-with-resources** closes ResultSet/Statement/Connection reliably
✅ **DAO interface** decouples DB layer from business layer
✅ **Service layer** owns validation + business rules
✅ **Custom exceptions** keep errors meaningful
✅ **Immutable model** reduces state bugs
✅ **Pagination** for list operations
✅ **Unique constraints** enforced in DB + optional early checks in service
✅ **Mapping function** to avoid repeated ResultSet parsing code