Spring JDBC

Title: Introduction to Spring JDBC

- In any application, **database usage is essential**, even if the application is not heavily data-driven.
- Most applications exist primarily to **store, retrieve, and process data**.
- To connect a Java application with a database, **JDBC (Java Database Connectivity)** is used.

Title: Challenges with Traditional JDBC

- Using **plain JDBC** requires writing many repetitive steps:
  - Loading the database driver
  - Creating a connection
  - Creating statements
  - Executing queries
  - Closing resources (connections, statements, result sets)
- These steps must be **manually managed** by the developer.
- Problems with traditional JDBC:
  - Time-consuming to write boilerplate code
  - Error-prone resource management
  - Developer must explicitly open and close connections

Title: Why Spring JDBC Is Needed

- Spring JDBC simplifies database interaction.
- It removes the need to manually manage:
  - Connections
  - Statements
  - Resource cleanup
- Spring JDBC provides built-in components to handle these repetitive tasks automatically.

Title: JDBC Template in Spring JDBC

- One of the most important components of Spring JDBC is **JdbcTemplate**.
- Responsibilities of JdbcTemplate:

    - Connects to the database
    - Executes SQL queries
    - Processes result sets
    - Returns output to the application
- It abstracts low-level JDBC code and simplifies database access.

Title: Database Connections and DataSource Concept

- In normal JDBC:
    - A new database connection is created for each request.
- This raises concerns:
    - Why create a new connection if existing ones are already available?
    - Applications may already have multiple open connections.
- **DataSource** helps solve this problem:
    - Manages database connections
    - Allows reuse of existing connections
- Spring JDBC provides DataSource support automatically using libraries.

Title: Spring JDBC Dependency Management

- When using Spring JDBC:
    - Required libraries are provided by Spring
    - No need to manually download dependencies
- Spring handles:
    - JDBC support
    - DataSource configuration
    - Connection management

Title: JDBC Drivers and Database Vendors

- JDBC provides **standard APIs**, but not implementations.
- Actual JDBC driver implementations are provided by **database vendors**.
- Different databases require different drivers:
    - PostgreSQL
    - MySQL
    - Oracle
    - H2
- The correct driver must be included based on the database being used.

Title: Using H2 Database with Spring JDBC

- In this module, the database used is **H2**.
- H2 is an **in-memory database**.
- Features of H2:
  - Lightweight
  - Easy to configure
  - Suitable for learning and testing
- Limitation:
  - Data is lost when the application stops (default in-memory behavior).
- When H2 dependency is added:
  - JDBC driver is automatically included.

Title: Scope of This Spring JDBC Module

- This module focuses on:
  - Using **Spring JDBC**
  - Working with **JdbcTemplate**
  - Connecting to a database using **H2**
- A new project will be created to:
  - Add Spring JDBC dependency
  - Add H2 database
  - Perform database operations using Spring JDBC

Title: Key Takeaways

- Traditional JDBC requires extensive boilerplate code.
- Spring JDBC simplifies database access and connection management.
- JdbcTemplate is the core component of Spring JDBC.
- DataSource enables efficient reuse of database connections.
- H2 is used as an in-memory database for this module.
- Spring provides JDBC drivers and required libraries automatically.

Understanding Spring JDBC and Database Connectivity

Why Databases Are Essential in Applications

- Almost every application requires a database to store information.
- Even applications that are not heavily data-driven still need to persist some form of data.
- Most applications rely entirely on data for their core functionality.

Using JDBC for Database Communication

- **JDBC (Java Database Connectivity)** is used to connect Java applications with databases.
- Traditional JDBC requires multiple manual steps:
    - Loading the JDBC driver
    - Opening a database connection
    - Creating statements
    - Executing queries
    - Closing the connection
- Managing these steps manually increases development effort and complexity.

Why Spring JDBC Is Needed

- **Spring JDBC** simplifies database operations by abstracting repetitive JDBC tasks.
- It reduces boilerplate code and manages:
    - Connection handling
    - Resource cleanup
    - Exception handling
- Spring JDBC provides multiple components, with **JdbcTemplate** being the most important.

JdbcTemplate and DataSource Concepts

JdbcTemplate

- A core Spring JDBC component.
- Helps with:
    - Connecting to the database
    - Executing SQL queries
    - Processing results
    - Returning output data
- Eliminates the need to write repetitive JDBC code.

DataSource and Connection Pooling

- Creating a new database connection for every request is inefficient.
- A **DataSource** allows reuse of existing connections.
- Spring JDBC provides:

    - Connection pooling
    - Efficient connection reuse
- Spring Boot automatically configures a **DataSource** when dependencies are added.

Database Driver and H2 In-Memory Database

JDBC Driver Responsibility

- JDBC APIs are part of the JDK, but actual implementations are provided by database vendors.
- Different DBMS (MySQL, Oracle, PostgreSQL, H2, etc.) require their own JDBC drivers.

Using H2 Database

- **H2** is an in-memory database.
- Features:
    - Stores data temporarily
    - Data is lost when the application stops
- Adding the H2 dependency automatically provides the JDBC driver.
- Suitable for learning and testing purposes.

Spring Boot Project Setup for Spring JDBC

Required Dependencies

- spring-boot-starter-jdbc
- h2

These two dependencies are sufficient to:

- Configure JDBC support
- Enable database connectivity

Project Validation Before Development

- Always run a newly downloaded project before adding logic.
- Ensures:
    - Application starts successfully
    - No configuration or dependency issues
- Successful execution confirms a stable starting point.

Student Management Use Case (Conceptual Focus)

Purpose of the Example

- The logic is not the focus.
- Goal is to understand:
  - Spring JDBC
  - Spring Boot integration
  - Layered architecture
- Example use case:
  - Add student
  - Retrieve student
  - Delete student

Mapping Database Tables to Java Classes

Relationship Between Table and Class

- A database table is conceptually similar to a Java class.
- Example:
  - Table: Student
  - Columns: rollNo, name, marks
- Java class:
  - Class name represents the table
  - Fields represent columns
- Each object of the class represents one row in the table.

Creating the Student Model Class

Student Fields

- rollNo
- name
- marks

Required Methods

- Getters and setters for all fields
- toString() method for object representation

Spring Annotations Used

- @Component
  - Enables Spring to manage the object lifecycle
- @Scope("prototype")
  - A new object is created each time it is requested
  - Required because multiple student objects are needed

Student Class Implementation

```java
@Component
@Scope("prototype")
public class Student {

    private int rollNo;
    private String name;
    private int marks;

    public int getRollNo() { return rollNo; }
    public void setRollNo(int rollNo) { this.rollNo = rollNo; }

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

    public int getMarks() { return marks; }
    public void setMarks(int marks) { this.marks = marks; }

    @Override
    public String toString() {
        return "Student{" +
                "rollNo=" + rollNo +
                ", name='" + name + '\'' +
                ", marks=" + marks +
                '}';
    }
}
```

Using ApplicationContext to Create Student Objects

Why Not Use new Keyword

- Objects should be created and managed by Spring.

- Enables dependency injection and lifecycle management.

Accessing Student Bean

```
ApplicationContext context =

SpringApplication.run(SpringbootjdbcdemoApplication.class,
args);

Student student = context×getBean(Student×class);
student.setRollNo(101);
student.setName("Sai");
student.setMarks(96);
```

Layered Architecture Decision

Why Not Direct Repository Access

- The application follows layered architecture.
- The main class acts as the client.
- Direct interaction with the repository is avoided.

Required Layers

- **Model** – Student entity
- **Service** – Business logic layer
- **Repository** – Database interaction layer

The service and repository layers will be created in the next module.

Key Takeaways

- Spring JDBC simplifies database access by removing manual JDBC management.
- JdbcTemplate and DataSource are central to Spring JDBC.
- H2 provides an in-memory database suitable for learning.
- Java classes map directly to database tables.
- Spring-managed prototype beans allow multiple entity instances.
- Clean layered architecture improves maintainability and design clarity.

Creating Service and Repository Layers in Spring JDBC

Objective

- Implement a layered architecture for a **student management system**:
  - **Service Layer:** Handles business logic and coordinates with repository.
  - **Repository Layer:** Handles data persistence (CRUD operations).
- Demonstrates the **separation of concerns** principle.

---

1. StudentService Class (Service Layer)

Purpose

- Acts as a mediator between the **client (main class)** and **repository layer**.
- Contains methods for business operations (e.g., adding a student, retrieving students).
- Does **not directly interact with the database**; delegates persistence tasks to the repository.

Implementation Details

- Belongs to service package.
- Annotated with @Service to mark it as a **Spring-managed bean**.
- Uses **dependency injection** to access the repository:
  - **Setter injection** used here with @Autowired.

Key Methods

1. **addStudent(Student s)**
   - Delegates storing the student object to the repository.
   - Example:

```
public void addStudent(Student s) {
    studentRepo.save(s);
}
```

2. **getStudents()**
    - Delegates fetching all students to the repository.
    - Follows **Spring Data JPA conventions** (findAll) for consistency.
    - Example:

```
public List<Student> getStudents() {
    return studentRepo.findAll();
}
```

---

2. StudentRepo Class (Repository Layer)

Purpose

- Handles all data persistence operations.
- Contains CRUD methods for interacting with the database.
- Belongs to repo package (can also be called dao).

Implementation Details

- Annotated with @Repository to mark it as a **Spring-managed persistence layer**.
- Currently uses **dummy implementations** for testing:
    - save(Student s) – Prints a message simulating database storage.
    - findAll() – Returns an empty list (placeholder for real database fetch).

Example Methods

```
@Repository
public class StudentRepo {

    public void save(Student s) {
        System.out.println("Student added successfully into
database...");
    }

    public List<Student> findAll() {
        List<Student> students = new ArrayList<>();
        return students;
    }
}
```

---

3. Main Application Flow

Steps in Main Class

1. Obtain **Student bean** from ApplicationContext.
2. Populate student details:

```
Student student = context.getBean(Student.class);
student.setRollNo(101);
student.setName("Sai");
student.setMarks(96);
```

3. Obtain **StudentService bean** from ApplicationContext.
4. Add the student via the service:

```
studentService.addStudent(student);
```

5. Retrieve and print all students via the service:

```
List<Student> students = studentService.getStudents();
System.out.println(students);
```

Key Notes

- Service layer **does not know** how the repository stores data.
- Repository layer is **solely responsible** for database operations.
- Current setup uses dummy data to validate flow before connecting to a real database (H2).

---

4. Design Principles Demonstrated

- **Separation of Concerns:** Service vs Repository responsibilities.
- **Dependency Injection:** Repository injected into Service using @Autowired.
- **Layered Architecture:** Main → Service → Repository.

- **Spring Stereotype Annotations:**
    - @Service → Service layer bean.
    - @Repository → Repository/persistence bean.

---

5. Next Steps

- Integrate **H2 database** with Spring JDBC.
- Use **JdbcTemplate** to store and fetch student data from the database.
- Replace dummy implementations in StudentRepo with real database operations.

Creating Database Schema and Initial Data Using H2

- When using traditional DBMS tools like **PostgreSQL** or **MySQL**, schemas are usually created directly through database clients.
- In this setup, **H2** is used without accessing the H2 web console.
- To initialize the database, schema and data are defined using SQL files inside the **resources** folder.

Using schema.sql for Table Creation

- A file named **schema.sql** is created in the resources directory.
- This file contains only the SQL required to define database structure.
- The CREATE TABLE statement is used to define the table.

Student Table Definition

- Table name: **student**
- Columns:
    - **rollNo**: int, marked as **primary key**
    - **name**: varchar(50)
    - **marks**: int

Schema SQL Example

```
create table student(
   rollNo int primary key,
   name varchar(50),
   marks int
);
```

Using data.sql for Preloading Data

- A separate file named **data.sql** is created to insert initial records.
- Data is inserted using standard **INSERT INTO** SQL statements.
- JSON files are not used for database initialization.

Insert Statement Structure

- Columns specified: rollNo, name, marks
- Values provided for each student record
- **Single quotes** are used for string values (mandatory in SQL)

Data SQL Example

```
insert into student(rollNo, name, marks) values(101, 'Sai', 96);
insert into student(rollNo, name, marks) values(102, 'Mahi', 92);
insert into student(rollNo, name, marks) values(103, 'Gopi', 84);
insert into student(rollNo, name, marks) values(104, 'Vijay', 73);
```

Automatic Execution by H2

- When the application starts:
  - **schema.sql** is executed first to create tables.
  - **data.sql** is executed next to insert predefined data.
- This behavior allows the database to be ready without manual setup.

Verifying Data Insertion

- After running the application, logs indicate:
  - Number of rows affected (e.g., 1 row affected).
- Successful execution implies that data has been stored in the database.
- Actual verification of stored data is done later using a **findAll** method.

Common Issues and Fixes

- **Column not found error**:
  - Caused by incorrect SQL syntax or mismatched column names.

- **Double quotes issue**:
  - String values must use **single quotes** in SQL.
  - Replacing double quotes with single quotes resolves execution errors.

Summary

- **schema.sql** defines database structure.
- **data.sql** preloads initial records.
- H2 automatically executes both files on startup.
- Proper SQL syntax (especially string quoting) is essential.
- Data insertion success is initially inferred from affected row count and later confirmed via retrieval operations.

Fetching Data from Database Using Spring JDBC and JdbcTemplate

Objective

- Retrieve records stored in the database using **Spring JDBC**.
- Implement data fetching logic in the **repository layer**.
- Use **JdbcTemplate.query()** with **RowMapper** to map database rows to Java objects.

---

1. Retrieving Data with SELECT Query

SQL Query for Fetching Data

- To retrieve all records from the table:

```
SELECT * FROM student
```

- This query fetches all rows from the student table.

---

2. Using JdbcTemplate.query() Method

Purpose

- Execute **SELECT queries**.
- Convert each row from the **ResultSet** into a Java object.

Method Signature

```
query(String sql, RowMapper<T> rowMapper)
```

Parameters

1. **SQL Query** – The SELECT statement.
2. **RowMapper** – Maps each row of the ResultSet to an object.

Return Type

- Returns a List<T> where T is the mapped object type (Student).

---

3. Understanding RowMapper

What is RowMapper

- An interface used to map **each row of the ResultSet** to a Java object.
- Called **once per row**.

Key Method

```
mapRow(ResultSet rs, int rowNum)
```

Parameters

- ResultSet rs → Contains data returned by the query.
- int rowNum → Row number (used internally; optional for logic).

Responsibility

- Extract column values from the ResultSet.
- Populate and return a **Student object**.

---

4. RowMapper Implementation (Anonymous Class)

```
RowMapper<Student> mapper = new RowMapper<Student>() {
    @Override
    public Student mapRow(ResultSet rs, int rowNum) throws
SQLException {
        Student student = new Student();
        student.setRollNo(rs.getInt("rollNo"));
        student.setName(rs.getString("name"));
        student.setMarks(rs.getInt("marks"));
        return student;
    }
};
```

Key Points

- rs.getInt() and rs.getString() retrieve column values.
- Column names must match database table columns.
- Each row becomes one Student object.

---

5. Fetching Data Using JdbcTemplate

```
String sql = "select * from student";
return jdbcTemplate.query(sql, mapper);
```

Flow

1. SQL query executed.
2. ResultSet generated.
3. RowMapper processes each row.
4. List of Student objects returned.

---

6. Using Lambda Expression with RowMapper

Why Lambda Works

- RowMapper is a **functional interface**.
- Allows concise lambda-based implementation.

Lambda-Based Implementation

```
public List<Student> findAll() {
    String sql = "select * from student";
    return jdbcTemplate.query(sql, (rs, rowNum) -> {
        Student student = new Student();
        student.setRollNo(rs.getInt("rollNo"));
        student.setName(rs.getString("name"));
        student.setMarks(rs.getInt("marks"));
        return student;
    });
}
```

Benefits

- Less boilerplate code.
- More readable and compact.
- Same behavior as anonymous class.

---

7. Execution Result

- Data successfully retrieved from database.
- Output includes a list of Student objects:

```
[Student{rollNo=101, name='Sai', marks=96},
 Student{rollNo=102, name='Mahi', marks=92},
 Student{rollNo=103, name='Gopi', marks=84}]
```

---

8. Key Takeaways

- **JdbcTemplate.query()** is used for SELECT operations.
- **RowMapper** converts ResultSet rows into objects.
- Lambda expressions simplify RowMapper implementation.
- Spring JDBC significantly reduces boilerplate compared to traditional JDBC.
- Repository layer handles database logic, keeping service clean.

Creating Database Schema and Initial Data Using H2

- When using traditional DBMS tools like **PostgreSQL** or **MySQL**, schemas are usually created directly through database clients.
- In this setup, **H2** is used without accessing the H2 web console.
- To initialize the database, schema and data are defined using SQL files inside the **resources** folder.

Using schema.sql for Table Creation

- A file named **schema.sql** is created in the resources directory.
- This file contains only the SQL required to define database structure.

- The CREATE TABLE statement is used to define the table.

Student Table Definition

- Table name: **student**
- Columns:
    - **rollNo**: int, marked as **primary key**
    - **name**: varchar(50)
    - **marks**: int

Schema SQL Example

```
create table student(
    rollNo int primary key,
    name varchar(50),
    marks int
);
```

Using data.sql for Preloading Data

- A separate file named **data.sql** is created to insert initial records.
- Data is inserted using standard **INSERT INTO** SQL statements.
- JSON files are not used for database initialization.

Insert Statement Structure

- Columns specified: rollNo, name, marks
- Values provided for each student record
- **Single quotes** are used for string values (mandatory in SQL)

Data SQL Example

```
insert into student(rollNo, name, marks) values(101, 'Sai', 96);
insert into student(rollNo, name, marks) values(102, 'Mahi', 92);
insert into student(rollNo, name, marks) values(103, 'Gopi', 84);
```

insert into student(rollNo, name, marks) values(104, 'Vijay', 73);

Automatic Execution by H2

- When the application starts:
  - **schema.sql** is executed first to create tables.
  - **data.sql** is executed next to insert predefined data.
- This behavior allows the database to be ready without manual setup.

Verifying Data Insertion

- After running the application, logs indicate:
  - Number of rows affected (e.g., 1 row affected).
- Successful execution implies that data has been stored in the database.
- Actual verification of stored data is done later using a **findAll** method.

Common Issues and Fixes

- **Column not found error**:
  - Caused by incorrect SQL syntax or mismatched column names.
- **Double quotes issue**:
  - String values must use **single quotes** in SQL.
  - Replacing double quotes with single quotes resolves execution errors.

Summary

- **schema.sql** defines database structure.
- **data.sql** preloads initial records.
- H2 automatically executes both files on startup.
- Proper SQL syntax (especially string quoting) is essential.
- Data insertion success is initially inferred from affected row count and later confirmed via retrieval operations.

Switching from H2 (In-Memory DB) to External Databases (PostgreSQL Example)

Overview: Why Switch from H2 to External DBMS

- **H2** is an **in-memory embedded database** used mainly for learning and testing.
- Data stored in H2 is lost when the application stops.
- Real-world applications typically use **external databases** such as **PostgreSQL**, **MySQL**, or **Oracle**.

- Spring JDBC allows switching databases **without changing repository, service, or main application code**.

Creating a Database and Table in PostgreSQL

- PostgreSQL is accessed using **pgAdmin**.
- A new database is created (example: Talisker).
- The existing schema.sql used for H2 is reused.

Table Creation (Executed in PostgreSQL):

```
create table student(
    rollNo int primary key,
    name varchar(50),
    marks int
);
```

Inserting and Verifying Data in PostgreSQL

- Data is manually inserted into PostgreSQL before running the Spring application.

Insert Statements:

```
insert into student(rollNo, name, marks) values(101, 'Sai',
96);
insert into student(rollNo, name, marks) values(102, 'Mahi',
92);
insert into student(rollNo, name, marks) values(103, 'Gopi',
84);
```

insert into student(rollNo, name, marks) values(104, 'Vijay', 73);

- Data can be verified using:
  - pgAdmin → View All Rows
  - or select * from student;

Updating Existing Data to Validate DB Switch

- Data is updated directly in PostgreSQL to confirm results are coming from Postgres and not H2.

Update Query Example:

```
update student set marks = 75 where rollNo = 102;
```

- Updated value (75) helps confirm the data source when fetched by the application.

Removing H2 Dependency and Adding PostgreSQL Driver

- **H2 dependency** is removed (or commented out) from pom.xml.
- PostgreSQL JDBC driver is added manually.

PostgreSQL Dependency:

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.7.8</version>
    <scope>compile</scope>
</dependency>
```

- Maven project is reloaded to download the driver.

Why Application Fails After Dependency Change

- Embedded databases (H2) auto-configure datasource properties.
- External databases **require explicit configuration**.
- Error occurs because **datasource URL is missing**.

Configuring External Database Using application.properties

- Configuration is added in application.properties.
- No changes are required in:
  - Repository classes
  - Service classes
  - Main application class

Required Properties:

- Database URL
- Username

- Password
- JDBC Driver class name

PostgreSQL Configuration Example:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/
mahichowdary
spring.datasource.username=postgres
spring.datasource.password=h@ppydays
spring.datasource.driver-class-name=org.postgresql.Driver
spring.application.name=springbootjdbcdemo
```

Understanding the JDBC URL Structure

- JDBC URLs always start with jdbc:
- Format:
  - jdbc:<dbms>://<host>:<port>/<database>
- PostgreSQL defaults:
  - Port: 5432
- MySQL default port (mentioned): 3306

Successful Application Execution with PostgreSQL

- Application starts successfully after configuration.
- Insert operations work (1 row affected).
- findAll() fetches data from PostgreSQL.
- Updated marks value (75) confirms PostgreSQL is the active database.

Key Takeaways

- Switching databases requires changes in only **two places**:
  1. **pom.xml** (JDBC driver dependency)
  2. **application.properties** (datasource configuration)
- Spring JDBC abstracts DB-specific logic.
- Same repository and query logic works across different DBMS.
- Database portability is achieved through configuration, not code changes.